Udacity SDC Nanodegree Semester 1: Computer Vision

Lesson 1: Course Intro

Overview of Program

The course is divided into robotics and deep learning approaches to self driving cars.

Projects

1. Lane tracking
2. Behavioural Cloning
3. Advanced Lane Finding
4. Sensor Fusion
5. Localisation
6. Controllers
7. Path Planning

Lesson 2: Computer Vision Fundamentals

Colors

Color is represented as RGB colour channels pixels.

<u>Steps to select lanes</u>
Apply thresholds to remove any colour that's not white and be left with white lanes.
Select region relative to vehicle frame (region masking)

This is not a robust algorithm.

Computer Vision

<u>**Canny Edge Detection**</u>
Computing the edge by calculating the gradient of an greyscale image.

<u>**Hough Transform**</u>

In image space, a line is plotted as x vs. y, but in 1962, Paul Hough devised a method for representing lines in parameter space, which we will call "Hough space" in his honor.
In Hough space, I can represent my "x vs. y" line as a point in "m vs. b" instead. The Hough Transform is just the conversion from image space to Hough space. <u>**So, the characterization of a line in image space will be a single point at the position (m, b) in Hough space.**</u>

Hough space is slope versus offset.

<u>**Hough Transform Properties**</u>

A point in image space describes a line in Hough space. So a line in an image is a point in Hough space and a point in an image is a line in Hough space

Two points in image space correspond to two lines in Hough Space. Not only that, but these lines must intersect

<u>This can also be accomplished in polar space (r, theta)</u>

Project 1

https://github.com/udacity/CarND-LaneLines-P1

Install mini conda
Submit Project as GitHub Repository

Setup anaconda environment

https://github.com/udacity/CarND-Term1-Starter-Kit/blob/master/README.md
https://github.com/udacity/CarND-Term1-Starter-Kit/blob/master/doc/configure_via_anaconda.md

Lesson 3: Intro to Neural Networks

Overview of Course

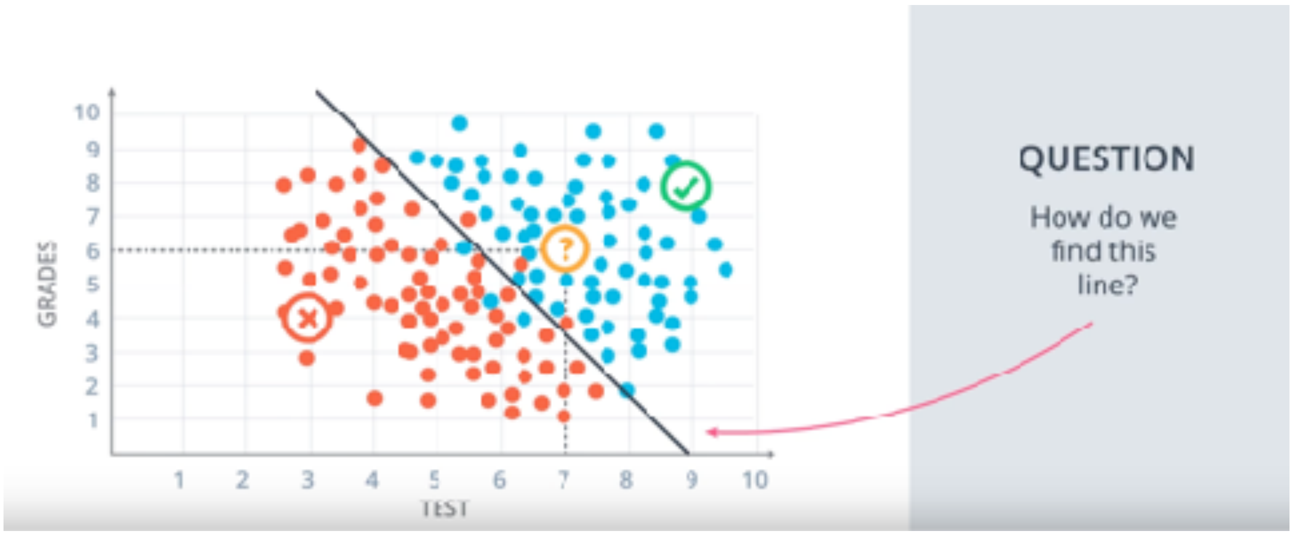
Deep learning has revolutionised computer vision. It has the potential to revolutionise AVs.

Machine Learning

Deep learning uses deep neural networks for machine learning.

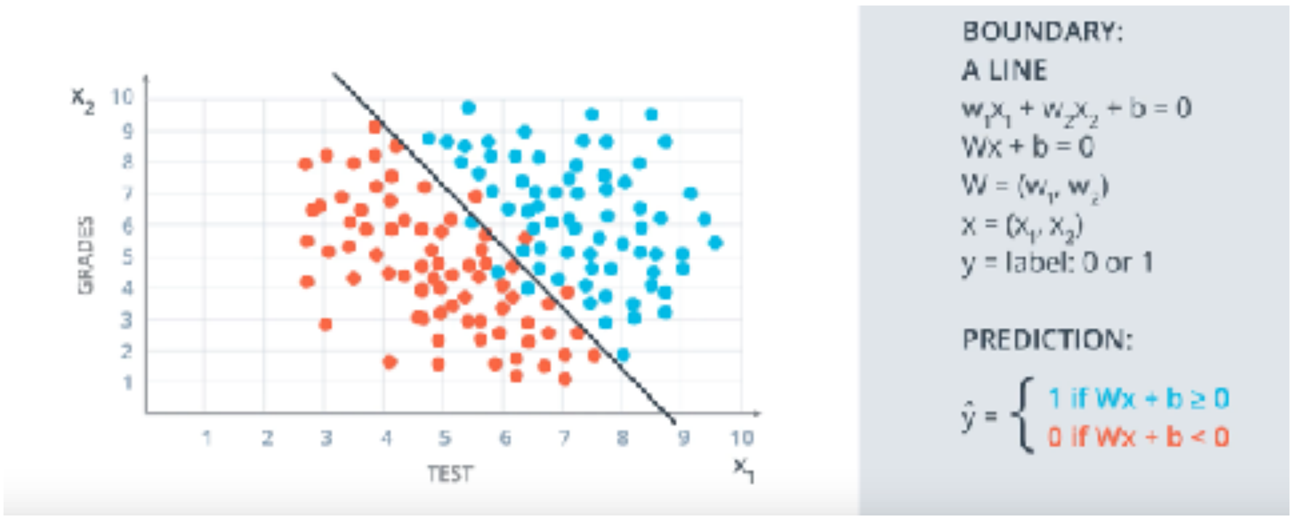
Linear Regression (i.e. least square fit) can be used to solve classification problems in a method called Logistic Regression.
Classification problems are important for self-driving cars. Self-driving cars might need to classify whether an object crossing the road is a car, pedestrian, and a bicycle. Or they might need to identify which type of traffic sign is coming up, or what a stop light is indicating.

Linear Boundaries



Solving classification problems between different datasets by separating datasets by lines

<u>**Wx + b = 0**</u>
x is input vector
y is label
y hat is predicted label
W is weights vector
b is bias



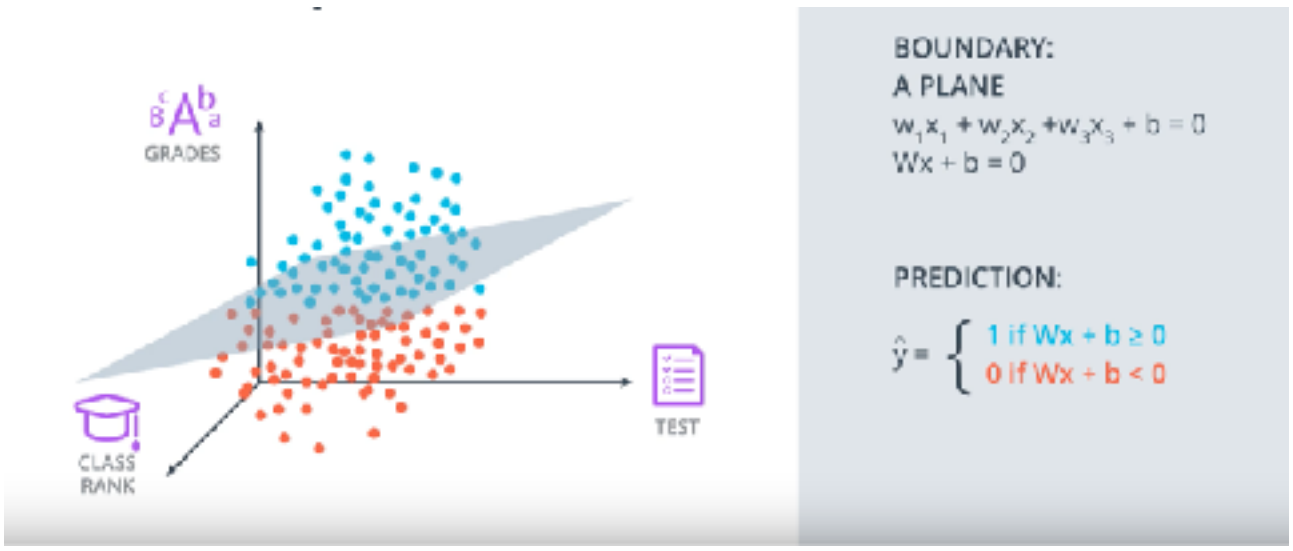
BOUNDARY:
A LINE
$w_1x_1 + w_2x_2 + b = 0$
$Wx + b = 0$
$W = (w_1, w_2)$
$x = (x_1, x_2)$
y = label: 0 or 1

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

Higher Dimensions

Fitting data with n columns. Data is n-dimensional states, Each data point has xn attributes and a single y-label

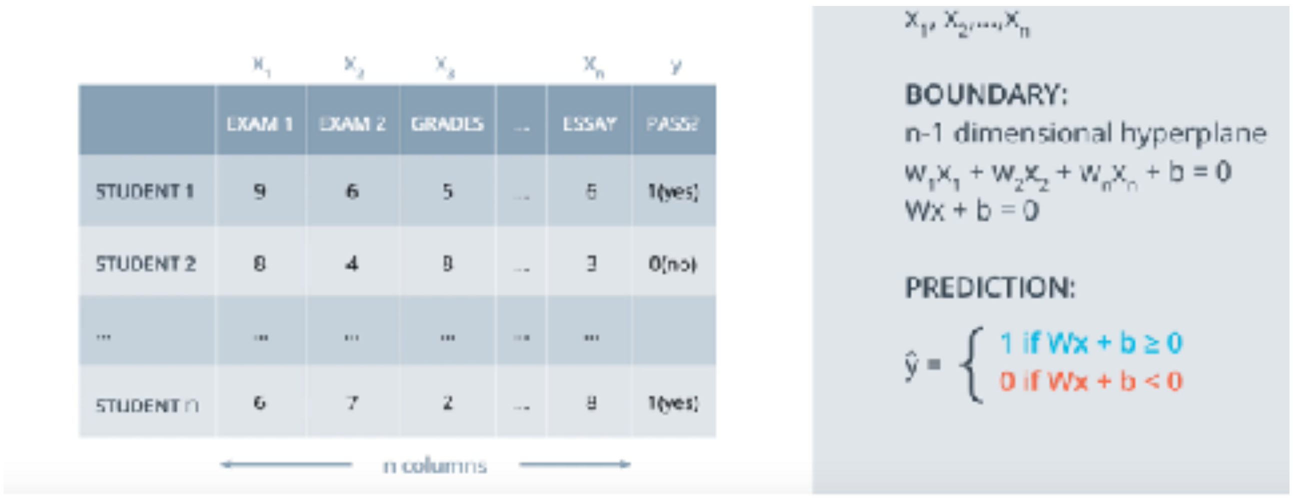


BOUNDARY:
A PLANE
$w_1x_1 + w_2x_2 + w_3x_3 + b = 0$
$Wx + b = 0$

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

For n-dimensional space,



	x_1	x_2	x_3		x_n	y
	EXAM 1	EXAM 2	GRADES	...	ESSAY	PASS?
STUDENT 1	9	6	5	...	5	1(yes)
STUDENT 2	8	4	8	...	3	0(no)
...	...	...	...	...	...	
STUDENT n	6	7	2	...	8	1(yes)

n columns

$x_1, x_2, \ldots, x_n$

BOUNDARY:
n-1 dimensional hyperplane
$w_1x_1 + w_2x_2 + w_nx_n + b = 0$
$Wx + b = 0$

PREDICTION:

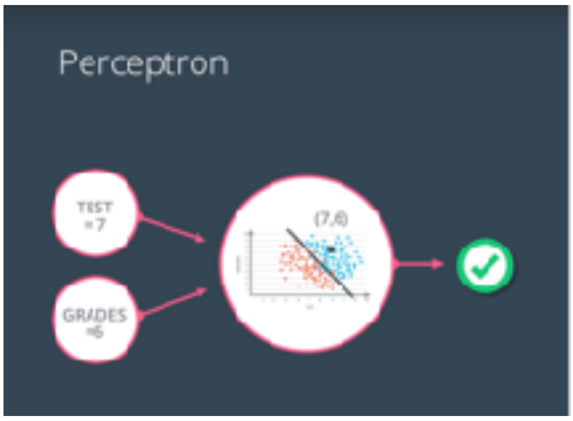
$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

W x + b = 0, where W is 1xn, X is nx1 and b is a single value.

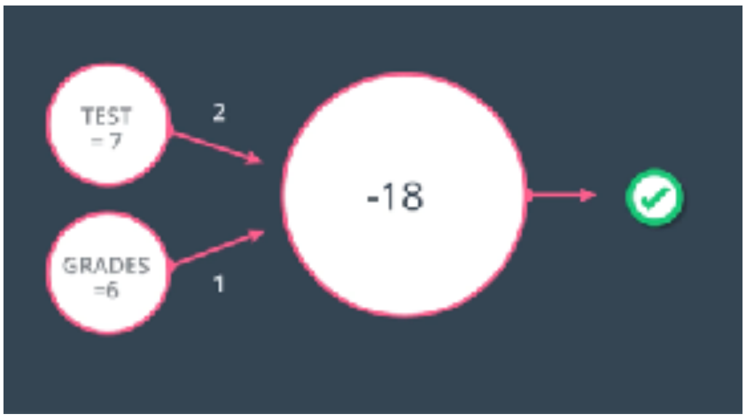
This describes a hyperplane which is a subspace whose dimension is one less than that of its ambient space. If a space is 3-dimensional then its hyperplanes are the 2-dimensional planes, while if the space is 2-dimensional, its hyperplanes are the 1-dimensional lines. This notion can be used in any general space in which the concept of the dimension of a subspace is defined.

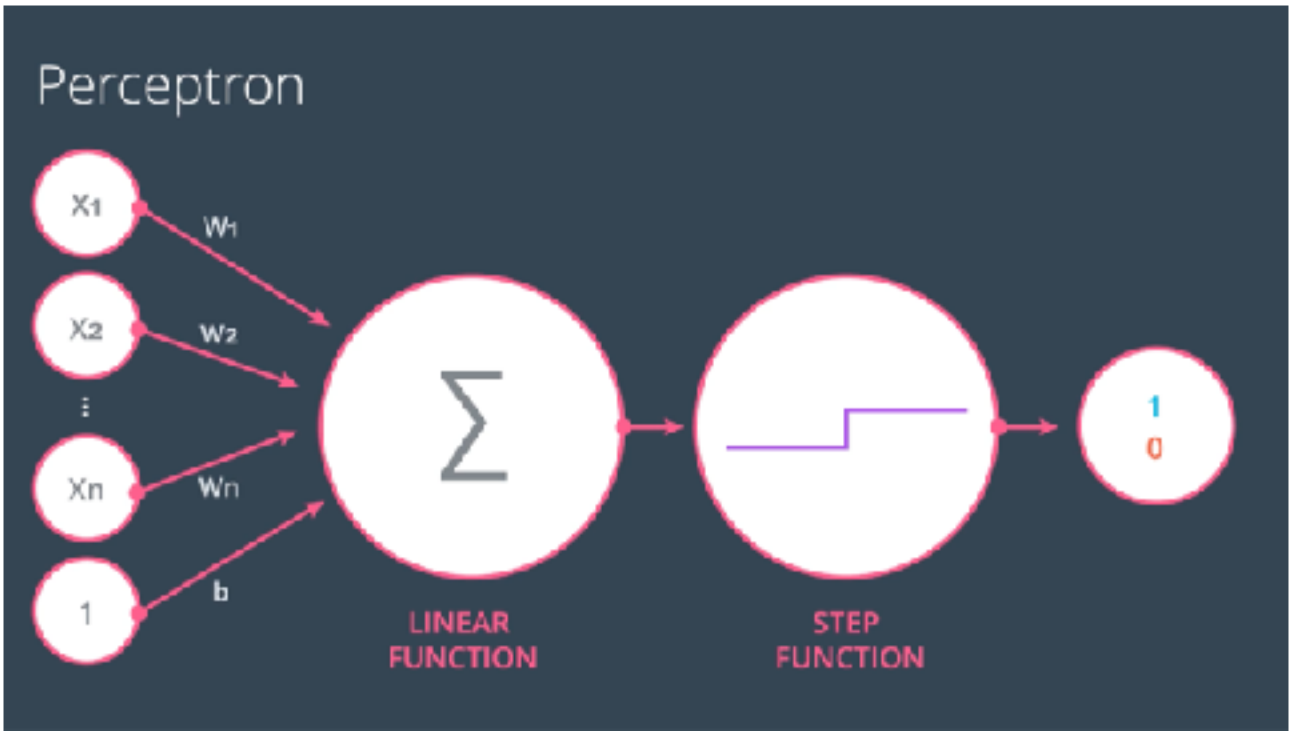
Perceptron

Building block of neural network (i.e. classifier). It is a node fitted with our data and linear boundary model. Given N inputs, it returns a label for the data point.



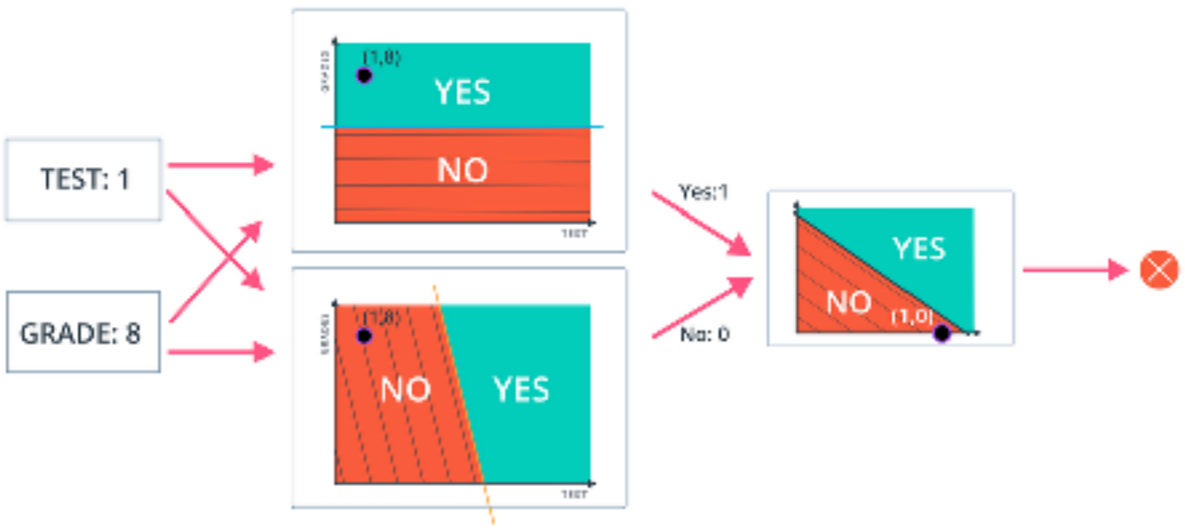
Labelling the graph using the weights (the edges) of the equation and bias labels the nodes





General case, N-inputs to a node with values (x1-xn) and edges with weights (w1-wn), and b corresponding to the bias unit. Finally the nodes calculates Wx+b to label the data point.

$$\sum_{i=1}^{m} w_i \cdot x_i$$



Data, like test scores and grades, are fed into a network of interconnected nodes. These individual nodes are called perceptrons, or artificial neurones, and they are the basic unit of a neural network. Each one looks at input data and decides how to categorise that data. In the example above, the input either passes a threshold for grades and test scores or doesn't, and so the two categories are: yes (passed the threshold) and no (didn't pass the threshold). These categories then combine to form a decision -- for example, if both nodes produce a "yes" output, then this student gains admission into the university.

Activation Functions

Finally, the result of the perceptron's summation is turned into an output signal! This is done by feeding the linear combination into an activation function.
Activation functions are functions that decide, given the inputs into the node, what should be the node's output? Because it's the activation function that decides the actual output, we often refer to the outputs of a layer as its "activations".
One of the simplest activation functions is the Heaviside step function. This function returns a 0 if the linear combination is less than 0. It returns a 1 if the linear combination is positive or equal to zero. The Heaviside step function is shown below, where h is the calculated linear combination:

$$f(h) = \begin{cases} 0 & \text{if } h < 0 \\ 1 & \text{if } h \geq 0 \end{cases}$$

Training Perceptrons (Neural Network Training)

Well, when we initialise a neural network, we don't know what information will be most important in making a decision. It's up to the neural network to learn for itself which data is most important and adjust how it considers that data.
It does this with something called weights.

When input comes into a perceptron, it gets multiplied by a weight value that is assigned to this particular input. For example, the perceptron above has two inputs, tests for test scores and grades, so it has two associated weights that can be adjusted individually. These weights start out as random values, and as the neural network network learns more about what kind of input data leads to a student being accepted into a university, the network adjusts the weights based on any errors in categorisation that results from the previous weights. This is called training the neural network.
A higher weight means the neural network considers that input more important than other inputs, and lower weight means that the data is considered less important. An extreme example would be if test scores had no affect at all on university acceptance; then the weight of the test score input would be zero and it would have no affect on the output of the perceptron.

Of course, with neural networks we won't know in advance what values to pick for biases. That’s ok, because just like the weights, the bias can also be updated and changed by the neural network during training. So after adding a bias, we now have a complete perceptron formula:

$$f(x_1, x_2, \ldots, x_m) = \begin{cases} 0 & \text{if } b + \sum w_i \cdot x_i < 0 \\ 1 & \text{if } b + \sum w_i \cdot x_i \geq 0 \end{cases}$$

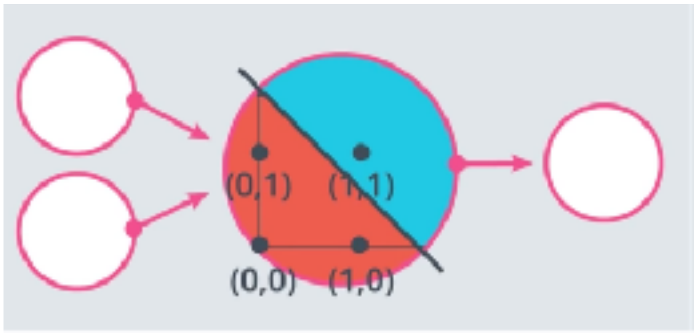
Perceptron Formula

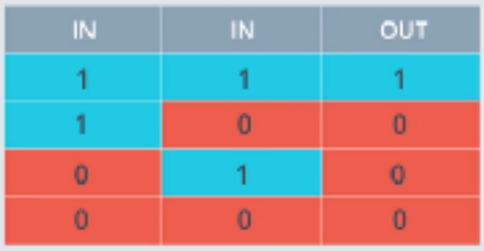
This formula returns 1 if the input (x1,x2,,xm)
belongs to the accepted-to-university category or returns 0.
The input is made up of one or more real numbers, each one represented by xi where m is the number of inputs & bias (b) are assigned a random value, and then they are updated using a learning algorithm like gradient descent. The weights and biases change so that the next training example is more accurately categorised, and patterns in data are "learned" by the neural network.

Perceptrons look like neurones in the brain.

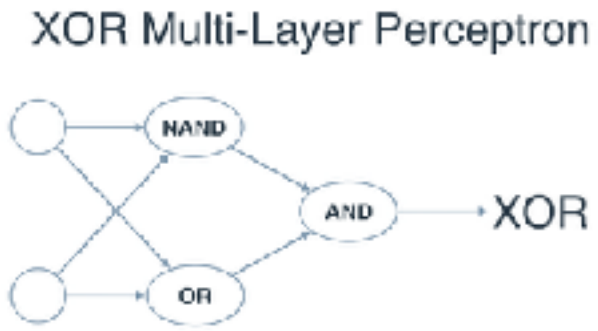
Neural networks are concatenated perceptions (i.e input from one is the output from another perceptions).

Perceptrons can be used to represent logical operators, e.g. AND operator



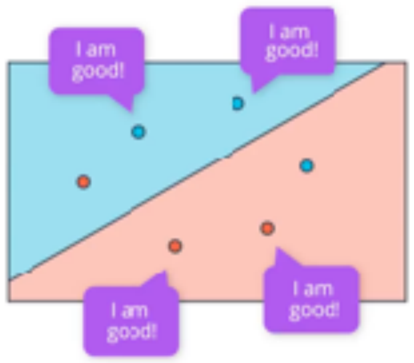


IN	IN	OUT
1	1	1
1	0	0
0	1	0
0	0	0

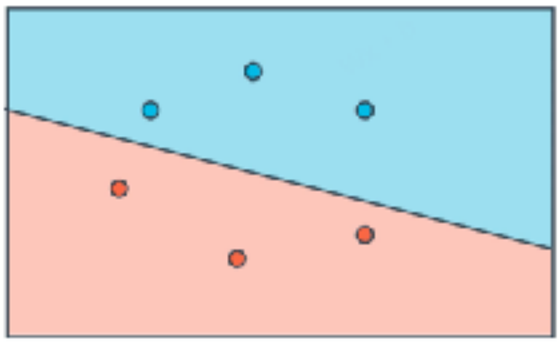


Perceptron Algorithm

How do we find the hyperplane parameters that correctly classifies the data ?

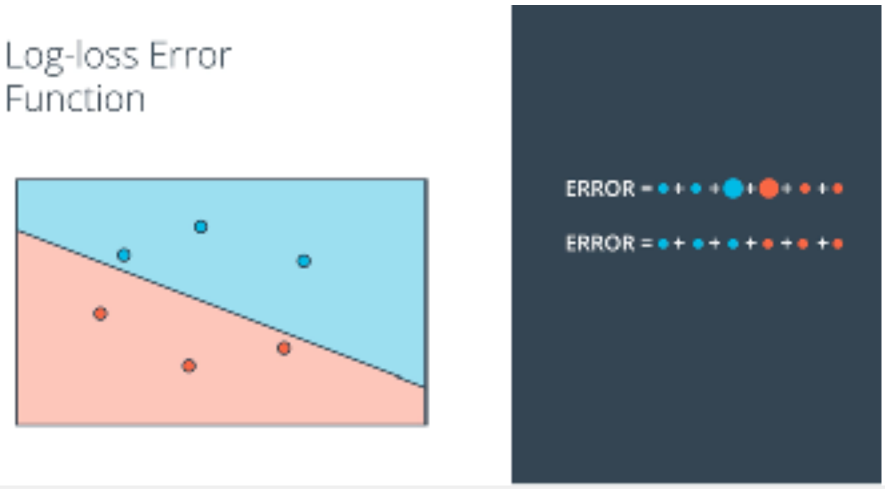


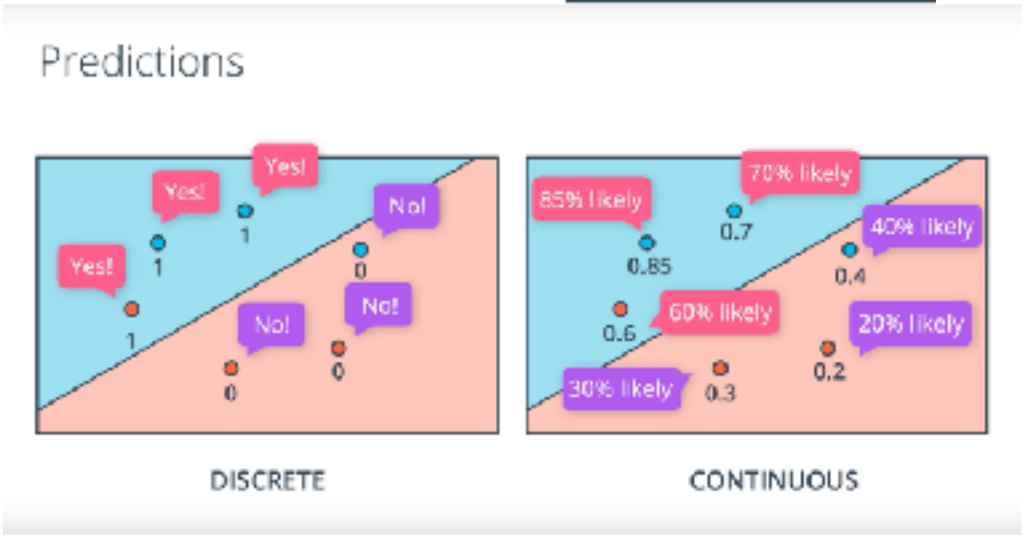
- Start by random weights.
- Evaluate current fit, using labelled data.
  - For each misclassified point (xi), we should inform the fit change. (Repeat for every misclassified point)
    - a misclassified point should move the line closer to it.
    - Learning rate is how much a line should move to each misclassified point (alpha)
    - We substrate false positive ((i.e if prediction =1)
      - wi + alpha* xi, b + alpha
    - Add False negative. (i.e if prediction =0)
      - wi - alpha* xi, b - alpha
- Repeat N times or till we reach a error threshold.



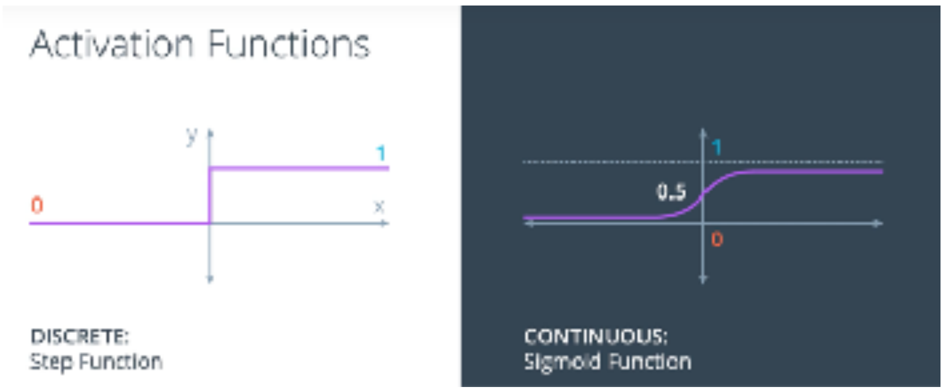
Nonlinear Regions

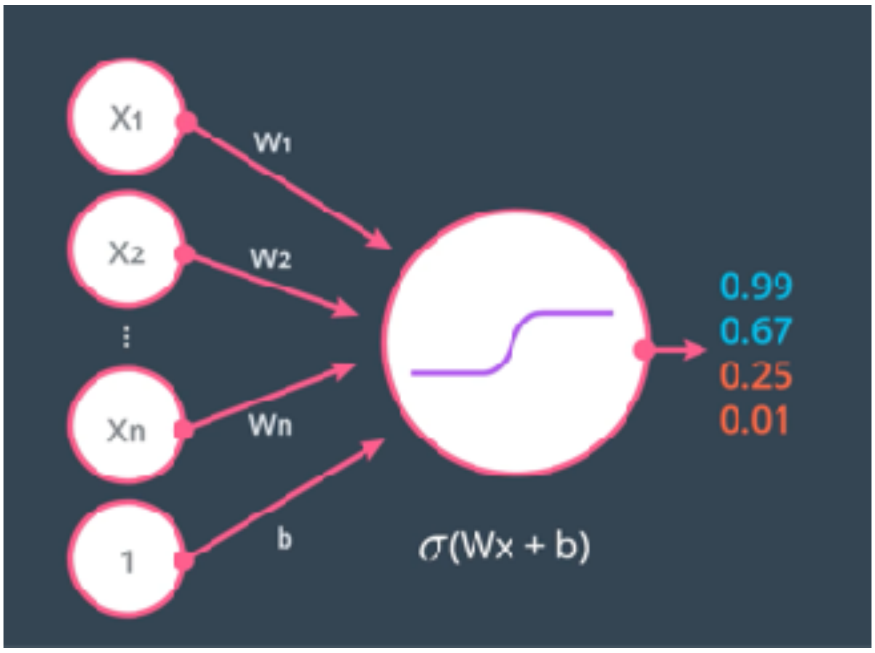
-Generalise preceptor algorithm to other types of curves.
-Error functions can be used.
-Using descent functions to decrease the error. (“Mount Errors”). Error functions must be a continuous functions (i.e. small variations in weights result in changes in errors).
- Log Loss Error (i.e.) Penalty based on distance on line for misclassification.





Using Sigmoid function instead of a step function enables us to create continuous predictions. For large positive (approach 1) and for large negative (approaches zero). This calculates the probability of a line being a certain classification by applying sigmoid to the output of Wx + b (prediction)





Multi-Class Classification and Softmax

For multi class classification,
First using a perceptron calculate a score for each class.
Calculate exponents for each score.
Normalise exponents.
(Calculate softmax for each score)

LINEAR FUNCTION SCORES:
$Z_1, \ldots, Z_n$

$$P(\text{class } i) = \frac{e^{Z_i}}{e^{Z_1} + \ldots + e^{Z_n}}$$

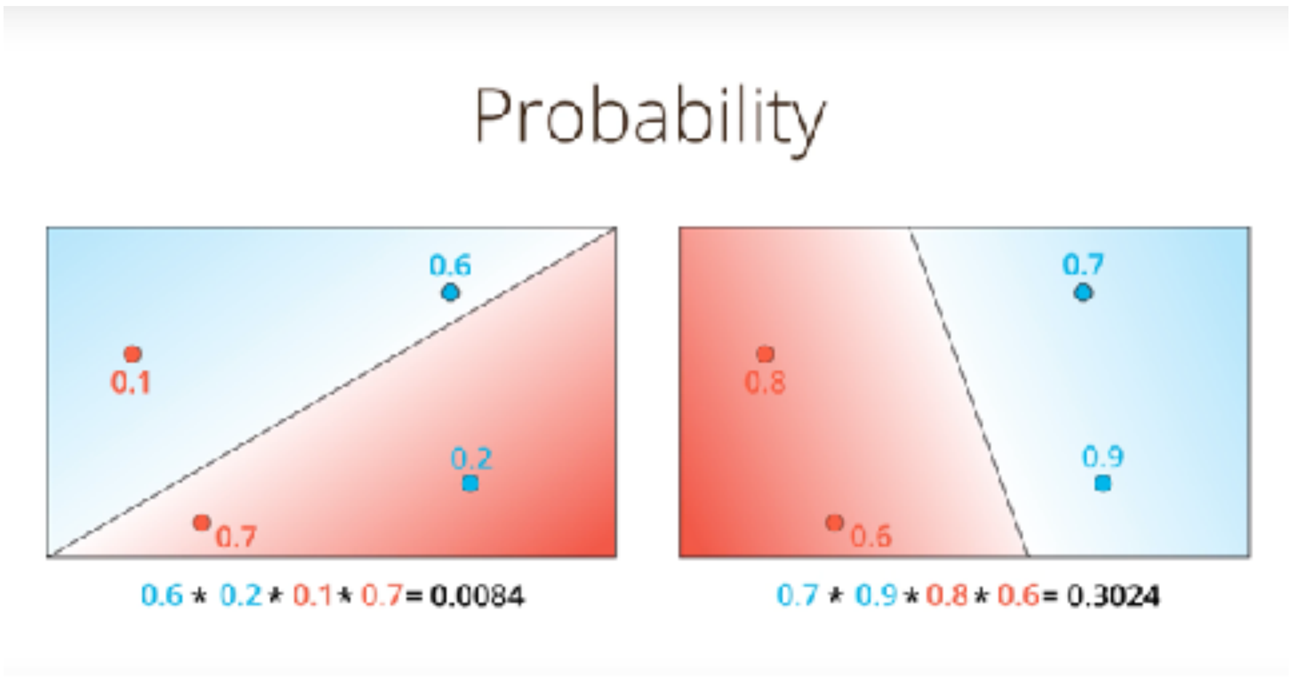
One-Hot Encoding

For non numerical data a binary truth table can be formed to represent different classes (or categories of data).

What we do instead is generate one boolean column for each category. Only one of these columns could take on the value 1 for each sample. Hence, the term one hot encoding.

Maximum Likelihood (ML)

The objective of the model is to maximise the probability of the prediction



A better model maximises the probabilities and minimises the error function

Maximising Probabilities
Cross Entropy is metric to measure total probability, using natural logarithms instead of product.
Cross Entropy is the sum of -ln(probabilities).
A good model has low cross entropy (i.e. higher probability, low error).

It can be thought as a measure of the fit of set events given a set of probabilities.

$$\text{Cross-Entropy} = -\sum_{i=1}^{m} y_i\ln(p_i) + (1-y_i)\ln(1-p_i)$$

Multi-Class Entropy
where n is the number of classes and m is the number of events that have occurred.

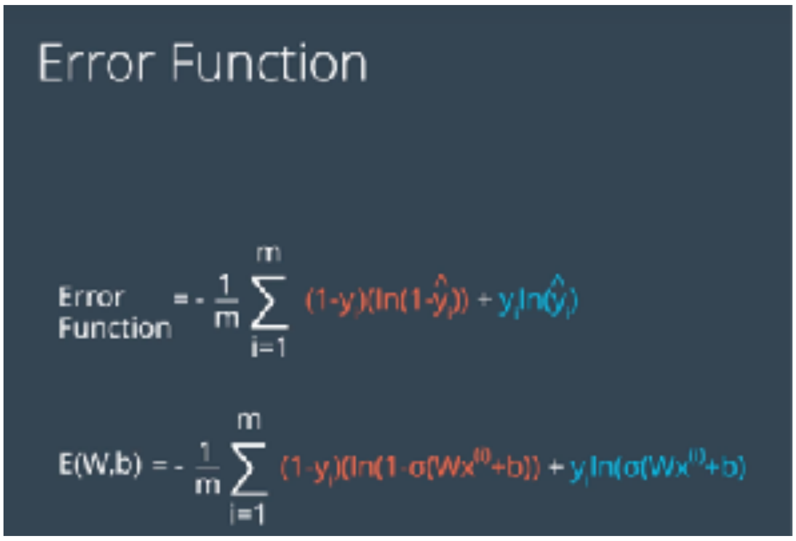
$$\text{Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\ \ln(p_{ij})$$

Logistic Regression

The Logistic Regression Algorithm, one of the most popular and useful algorithms in Machine Learning, and the building block of all that constitutes Deep Learning. And it basically goes like this:

- Take your data
- Pick a random model
- Calculate the error
- Minimize the error, and obtain a better model (using Gradient Descent)
- Enjoy!

Error function= Average of CE.
Since the prediction is simply the sigmoid of Wx+b
The error function can be expressed accordingly,

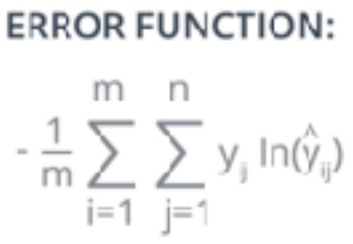


Error Function

$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)(\ln(1-\hat{y}_i)) + y_i\ln(\hat{y}_i)$$

$$E(W,b) = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)(\ln(1-\sigma(Wx^{(i)}+b)) + y_i\ln(\sigma(Wx^{(i)}+b))$$

For multi-class prediction,

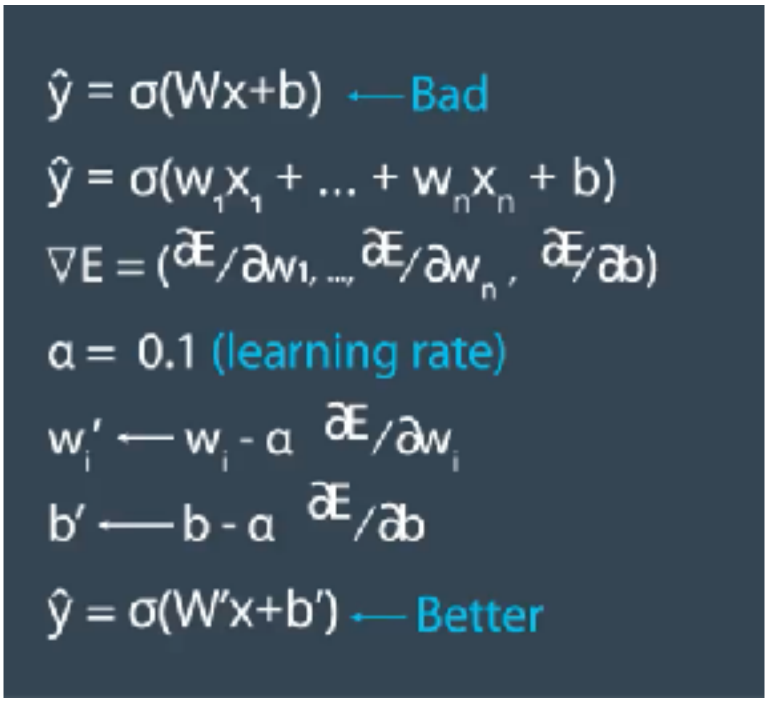


ERROR FUNCTION:

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_j \ln(\hat{y}_{ij})$$

Gradient Descent

Follow the negative gradient of the Error function



$\hat{y} = \sigma(Wx+b) \longleftarrow$ Bad

$\hat{y} = \sigma(w_1x_1 + \ldots + w_nx_n + b)$

$\nabla E = (\partial E/\partial w_1, \ldots, \partial E/\partial w_n, \partial E/\partial b)$

$\alpha = 0.1$ (learning rate)

$w_i' \longleftarrow w_i - \alpha\, \partial E/\partial w_i$

$b' \longleftarrow b - \alpha\, \partial E/\partial b$

$\hat{y} = \sigma(W'x+b') \longleftarrow$ Better

The gradient is the partial derivatives of the Error function relative to the Weights, and Bias. Which is the difference between the gradient and prediction times the scalar coordinates of the point. So, a small gradient means we'll change our coordinates by a little bit, and a large gradient means we'll change our coordinates by a lot.

$$\nabla E = -(y - \hat{y})(x_1, \ldots, x_n, 1).$$

$w_i' \leftarrow w_i + \alpha(y - \hat{y})x_i.$

Similarly, it updates the bias in the following way:

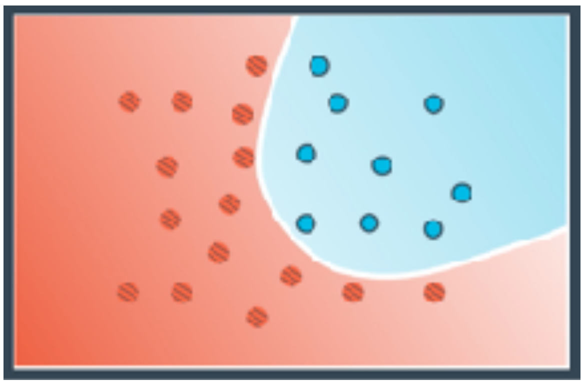
$b' \leftarrow b + \alpha(y - \hat{y}).$

Where, alpha is the learning rate divide by the number of events (or points).

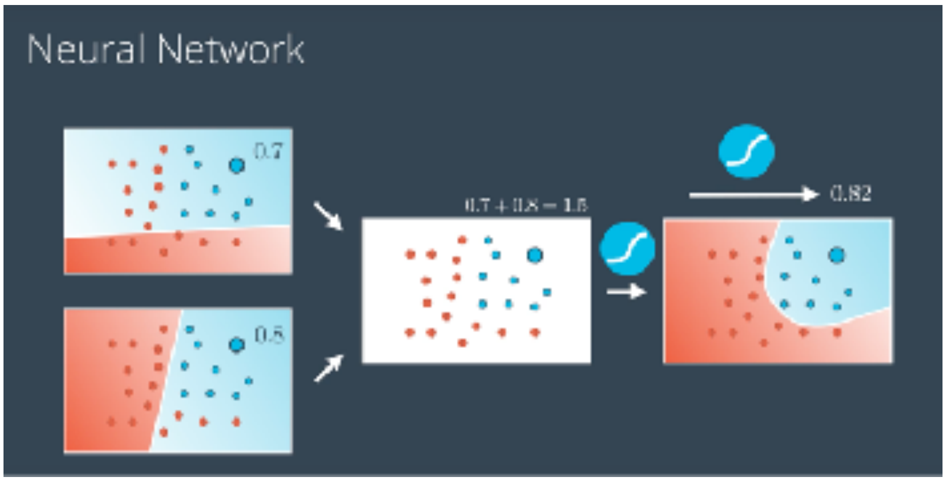
Unlike the perceptron algorithm, gradient descent uses both correctly classified and incorrectly classified to maximise likelihood (in continuous space).

Non linear Models

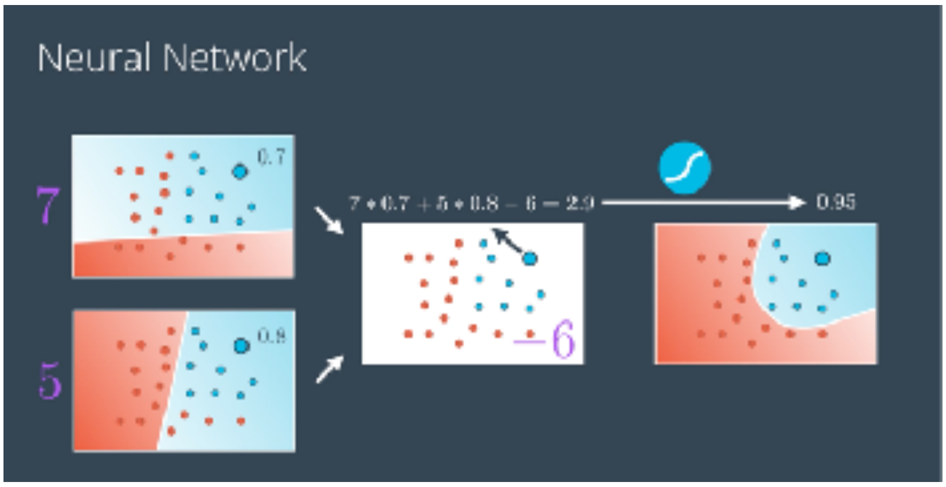
In some cases, datasets are not linearly separable.



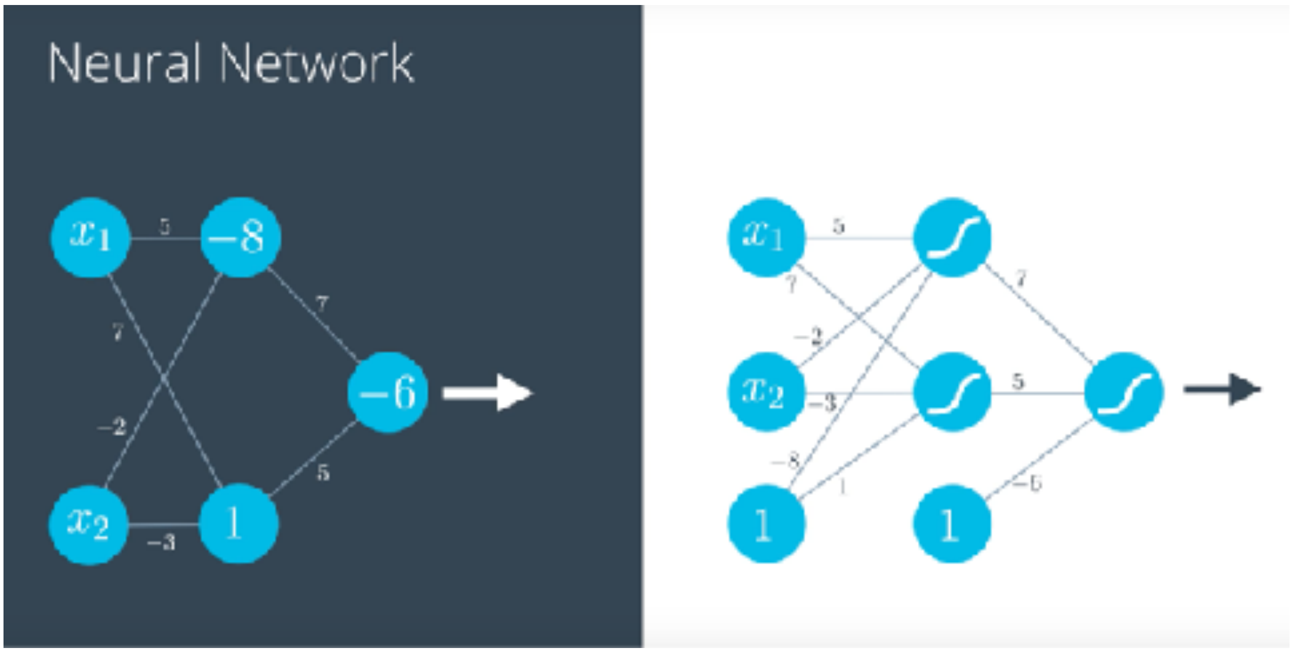
This is where neural networks come into play (i.e Multilayer Perceptrons) by combining two linear models (as if two models are superimpose).
The process for combining models. Add the probabilities from both models and use sigmoid to bound from 0 to 1.



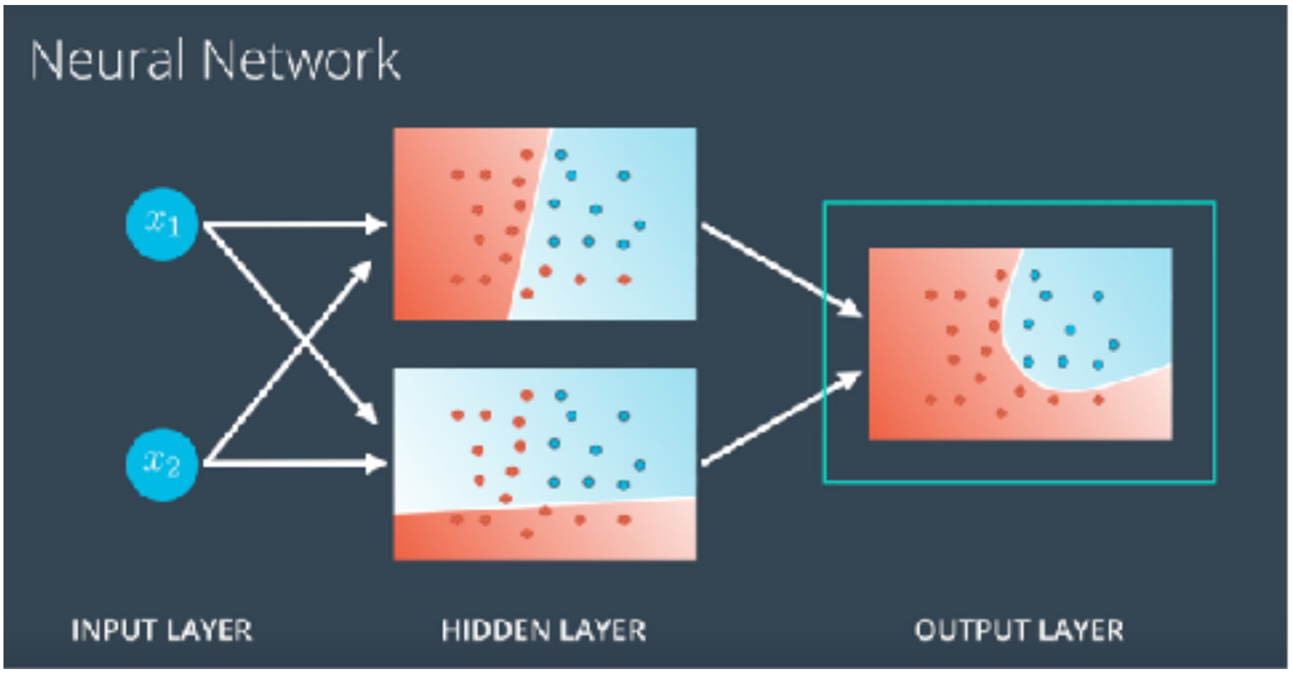
Neural networks are built using linear combinations of models (using weights & biases).



Using the node notation, this can be represented as such



Multiple layers

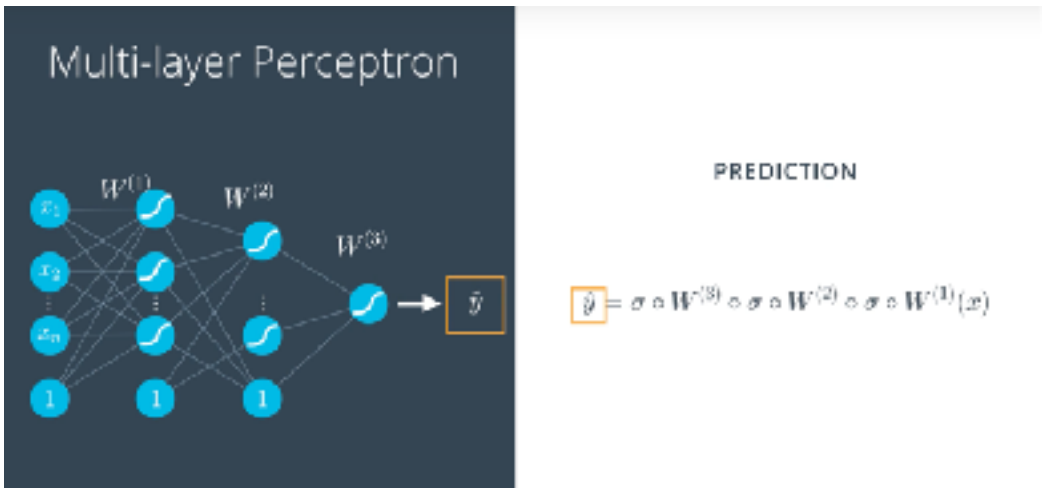


For Neural Networks, The number of output layers is equivalent to the number of classes being classified (multi class not binary classification). Then using softmax, one can determine which class is more probable.

Deep Neural Networks, Have multiple hidden layers.

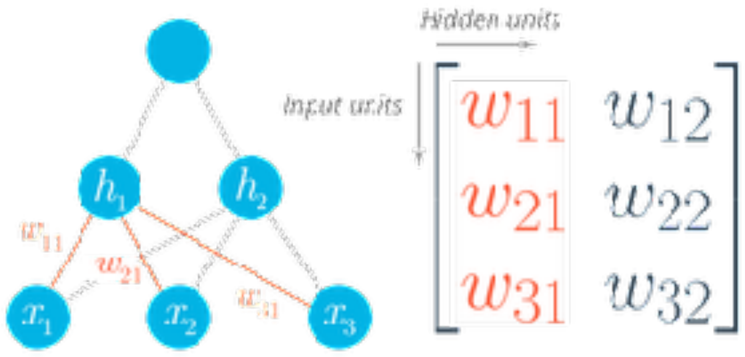
Feedforward

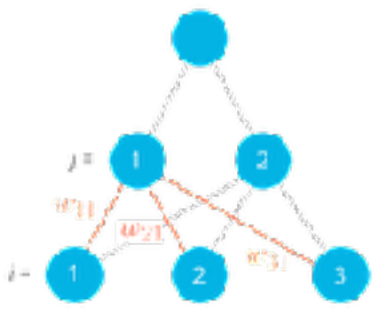
Feedforward is the process neural networks use to turn the input into an output. It is a series of sigmoid function matrices (weights) applied to a input vector.


Multi-layer Perceptron
$W^{(1)}$
$W^{(2)}$
$W^{(3)}$
x_1
x_2
x_n
1
$\hat{y}$
PREDICTION
$\hat{y} = \sigma \circ W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$

Multilayer Perceptrons

Before, we were able to write the weights as an array, indexed as wi
But now, the weights need to be stored in a matrix, indexed as Wij. Each row in the matrix will correspond to the weights leading out of a single input unit, and each column will correspond to the weights leading in to a single hidden unit. For our three input units and two hidden units, the weights matrix looks like this:


Hidden units
Input units
h_1
h_2
x_1
x_2
x_3
w_{11}
w_{21}
w_{31}
$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$



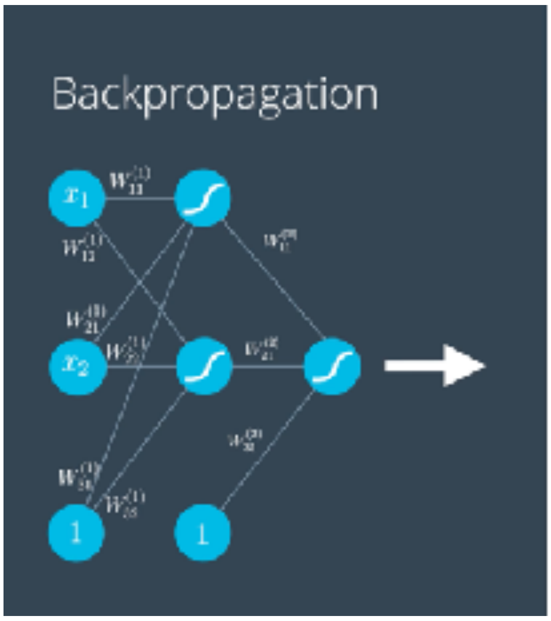
$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Training NN using Backpropagation

Training a neural network the method known as backpropagation is employed. In a nutshell, backpropagation will consist of:

- Doing a feedforward operation.
- Comparing the output of the model with the desired output.
- Calculating the error.
- Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights.
- Use this to update the weights, and get a better model.
- Continue this until we have a model that is good.

Formal

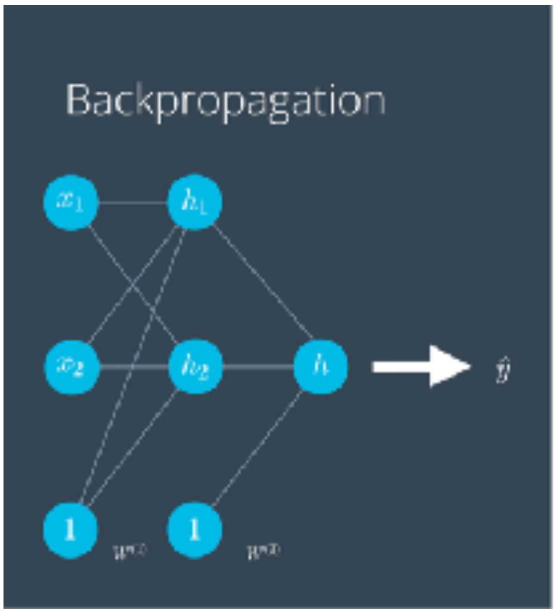


$$\hat{y} = \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

$$W^{(1)} = \begin{pmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} W_{11}^{(2)} \\ W_{21}^{(2)} \\ W_{31}^{(2)} \end{pmatrix}$$

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial W_{11}^{(1)}} & \frac{\partial E}{\partial W_{12}^{(1)}} & \frac{\partial E}{\partial W_{11}^{(2)}} \\ \frac{\partial E}{\partial W_{21}^{(1)}} & \frac{\partial E}{\partial W_{22}^{(1)}} & \frac{\partial E}{\partial W_{21}^{(2)}} \\ \frac{\partial E}{\partial W_{31}^{(1)}} & \frac{\partial E}{\partial W_{32}^{(1)}} & \frac{\partial E}{\partial W_{31}^{(2)}} \end{pmatrix}$$

$$W_{ij}^{(k)} \leftarrow W_{ij}^{(k)} - \alpha \frac{\partial E}{\partial W_{ij}^{(k)}}$$



$$E(W) = -\frac{1}{m} \sum_{i=1}^{m} y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i)$$

$$E(W) = E(W_{11}^{(1)}, W_{12}^{(1)}, \ldots, W_{31}^{(2)})$$

$$\nabla E = \left(\frac{\partial E}{\partial W_{11}^{(1)}}, \cdots, \frac{\partial E}{\partial W_{31}^{(2)}}\right)$$

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}$$

Using the sigmoid derivative properties, we can easily calculate all the required partial derivatives as shown below,

$$h = W_{11}^{(2)}\sigma(h_1) + W_{21}^{(2)}\sigma(h_2) + W_{31}^{(2)}$$

$$\frac{\partial h}{\partial h_1} = W_{11}^{(2)}\sigma(h_1)[1 - \sigma(h_1)]$$

Lesson 4: Mini Flow

This is a lab to build understanding of back propagation and differentiable graphs.

Mini Flow architecture

A generic node

- Inbound nodes,
- Outbound nodes
- Value.
- Forward propagation x
- Backward propagation method.

Forward Propagation

In order to define your network, you'll need to define the order of operations for your nodes. Given that the input to some node depends on the outputs of others, you need to flatten the graph in such a way where all the input dependencies for each node are resolved before trying to run its calculation. This is a technique called a topological sort. Topological sorting using Kahn's Algorithm.

Linear Algebra in Neural Networks

Linear algebra nicely reflects the idea of transforming values between layers in a graph. In fact, the concept of a transform does exactly what a layer should do - it converts inputs to outputs in many dimensions.
Let's go back to our equation for the output.

$$o = \sum_i x_i w_i + b$$

Equation (1)

Consider a Linear node with 1 input and k outputs (mapping 1 input to k outputs). In this context an input/output is synonymous with a feature.
In this case

$$X = \begin{bmatrix} X_{11} \end{bmatrix}$$

1 by 1 matrix, with 1 element. The subscript represents the row (1) and column (1) of the element in the matrix.
W becomes a 1 by k matrix (looks like a row).

$$W = \begin{bmatrix} W_{11} & W_{12} & W_{13} & \dots & W_{1k} \end{bmatrix}$$

A 1 by k weights row matrix.

The result of the matrix multiplication of X andW is a 1 by k matrix. Since b is also a 1 by k row matrix (1 bias per output), b is added to the output of the
What if we are mapping n inputs to k outputs?
Then X is now a 1 by n matrix and W is a n by k matrix. The result of the matrix multiplication is still a 1 by k matrix so the use of the biases remain the same.

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} & \dots & X_{1n} \end{bmatrix}$$

X is now a 1 by n matrix, n inputs/features.

$$W = \begin{bmatrix} W_{11} & W_{12} & W_{13} & \dots & W_{1k} \\ W_{21} & W_{22} & W_{23} & \dots & W_{2k} \\ W_{31} & W_{32} & W_{33} & \dots & W_{3k} \\ \vdots & & & \ddots & \vdots \\ W_{n1} & W_{n2} & W_{n3} & \dots & W_{nk} \end{bmatrix}$$

W is now a n by k matrix - input features by number of outputs.

$$b = \begin{pmatrix} b_1 & b_2 & b_3 & \dots & b_k \end{pmatrix}$$

Row matrix of biases, one for each output.

Let's take a look at an example of n input features. Consider an 28px by 28px greyscale image, as is in the case of images in the MNIST dataset. We can reshape (flatten) the image such that it's a 1 by 784 matrix, where n = 784. Each pixel is an input/feature. Here's an animated example emphasizing a pixel is a feature.

Batch Processing NN

In practice, it's common to feed in multiple data examples in each forward pass rather than just 1. The reasoning for this is the examples can be processed in parallel, resulting in big performance gains. The number of examples is called the batch size. Common numbers for the batch size are 32, 64, 128, 256, 512. Generally, it's the most we can comfortably fit in memory.

X becomes a m by n matrix (**where m is the batch size, by the number of input features, n**) and **W and b remain the same.** The result of the matrix multiplication is now m by k (batch size by number of outputs), so the addition of b is broadcast over each row.

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} & \dots & X_{1n} \\ X_{21} & X_{22} & X_{23} & \dots & X_{2n} \\ X_{31} & X_{32} & X_{33} & \dots & X_{3n} \\ \vdots & & & \ddots & \vdots \\ X_{m1} & X_{m2} & X_{m3} & \dots & X_{mn} \end{bmatrix}$$

X is now a m by n matrix. Each row (one example from the batch) has n inputs/features.

Equation (1) turns into:

$$Z = XW + b$$

Equation (2) can also be viewed as
Z=XW+B where Bi s the biases vector, b, stacked m times. Due to broadcasting it's abbreviated to Z = X W + b.

Sigmoid

Apply sigmoid function to output i.e. activation function instead of a binary step function

Gradient Descent

Simple formula, adjust the parameters to minimise cost function.
x = x - learning_rate * gradient_of_x

The insight of GD is optimising the learning rate to minimise cost.

<u>In order to figure out how we should alter a parameter to minimise the cost, we must first find out what effect that parameter has on the cost.</u>

Hence, Backpropagation

We simply calculate the derivative of the cost with respect to each parameter in the network. The gradient is a vector of all these derivatives.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a version of Gradient Descent where on each forward pass a batch of data is randomly sampled from total dataset. Remember when we talked about the batch size earlier? That's the size of the batch. Ideally, the entire dataset would be fed into the neural network on each forward pass, but in practice, it's not practical due to memory constraints. SGD is an approximation of Gradient Descent, the more batches processed by the neural network, the better the approximation.
A naïve implementation of SGD involves:

1. Randomly sample a batch of data from the total dataset.
2. Running the network forward and backward to calculate the gradient (with data from (1)).
3. Apply the gradient descent update.
4. Repeat steps 1-3 until convergence or the loop is stopped by another mechanism (i.e. the number of epochs).

If all goes well, the network's loss should generally trend downwards, indicating more useful weights and biases over time.

$$\boxed{X = X - \nabla C \alpha}$$

X: Trainable

∇C: Gradient

α: learningrate.

$$\boxed{\begin{tikzcd} X \arrow[r] & W \arrow[r] & \text{Activation} \\ g \arrow[r] & w \arrow[r] & X. \end{tikzcd}}$$

$$\frac{\delta G}{\delta X} = \underset{\text{gradient}}{G} . \underset{\text{weight}}{W}^T$$

$$\frac{\delta G}{\delta W} = X^T . G$$

input,

$X =$ datasets x Features

$W =$ Features x Output

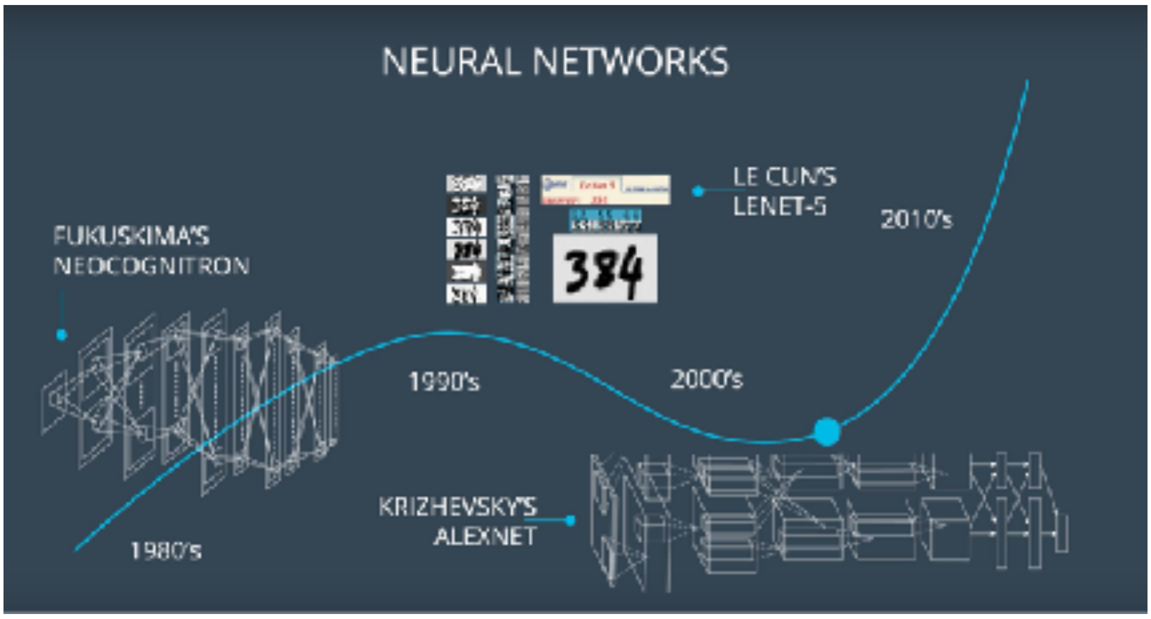
$G =$ Cost per data point

Lesson 5: TensorFlow

Requirements

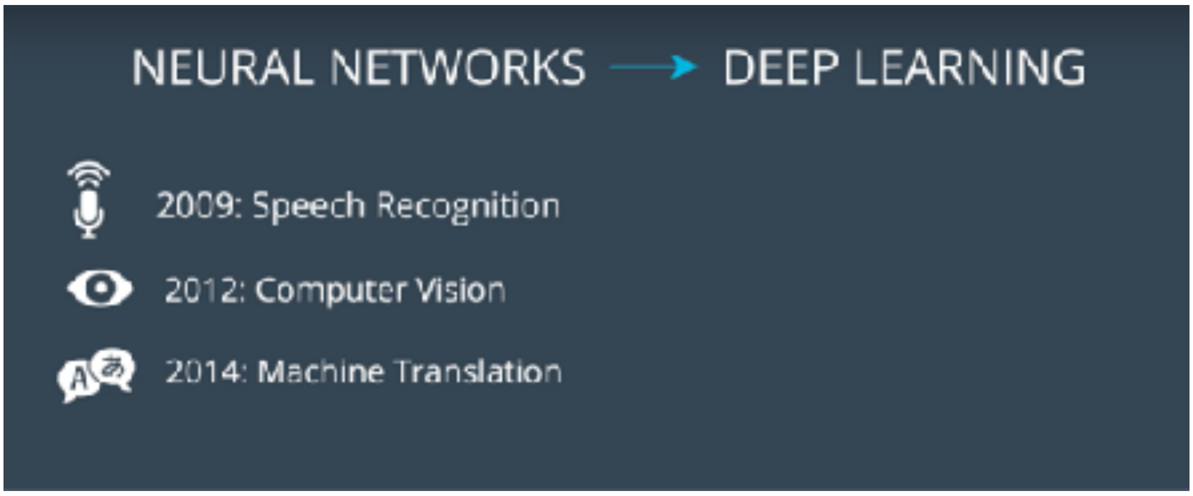
Python 3.4 and Anaconda

A Primer on NN



Research on NN was done since 1980s, but was limited by datasets and computing power.

Recent Applications are:



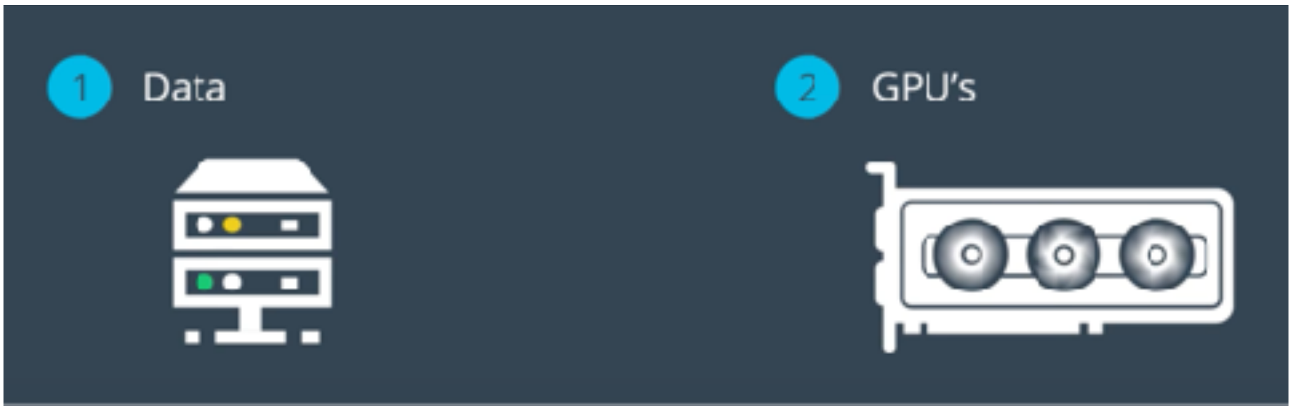
NEURAL NETWORKS → DEEP LEARNING

2009: Speech Recognition

2012: Computer Vision

2014: Machine Translation

Thanks to



Introduction to TF

All data is stored as an object called tensor.

tensors support many types such as int32, floats, bools, etc.
tensors can be n-dimensional

```
# A is a 0-dimensional int32 tensor
A = tf.constant(1234)
# B is a 1-dimensional int32 tensor
B = tf.constant([123,456,789])
# C is a 2-dimensional int32 tensor
C = tf.constant([ [123,456,789], [222,333,444] ])
```

TensorFlow's api is built around the idea of a computational graph, a session is used to run a graph.

tf.placeholder()
Sadly you can't just set x to your dataset and put it in TensorFlow, because over time you'll want your TensorFlow model to take in different datasets with different parameters. You need tf.placeholder()!
tf.placeholder() returns a tensor that gets its value from data passed to the tf.session.run() function, allowing you to set the input right before the session runs.

```
x = tf.placeholder(tf.string)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Hello World'})
```

Multiple Inputs

```
x = tf.placeholder(tf.string)
y = tf.placeholder(tf.int32)
z = tf.placeholder(tf.float32)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Test String', y: 123, z: 45.67})
```

```
Example
def run():
    output = None
    logit_data = [2.0, 1.0, 0.1]
    logits = tf.placeholder(tf.float32)

    softmax = tf.nn.softmax(logits)

    with tf.Session() as sess:
        output = sess.run(softmax, feed_dict={logits: logit_data})

    return output
```

Variables

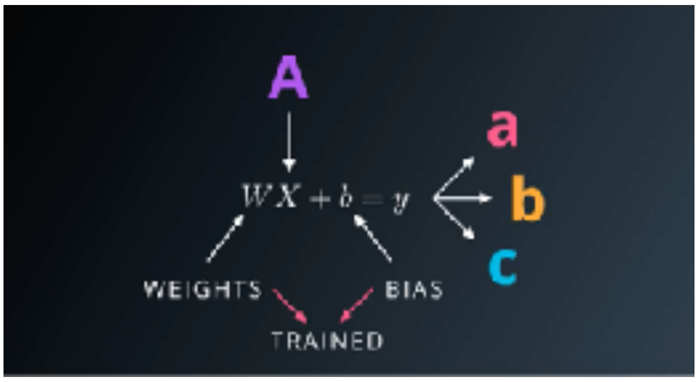
https://www.tensorflow.org/programmers_guide/variables

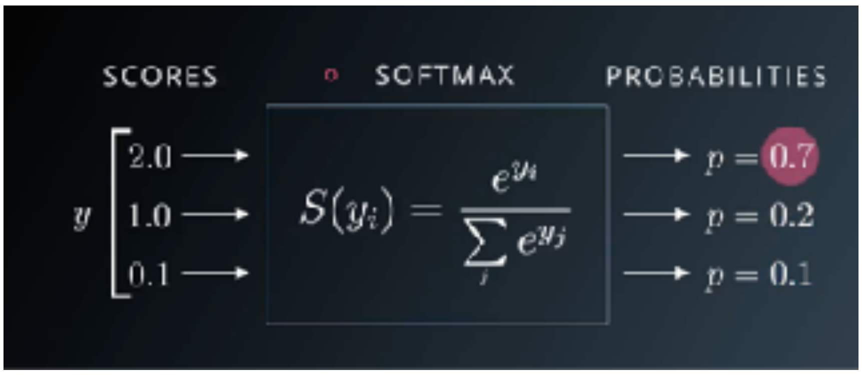
Classification

In particular, supervised classification. This is the central building block of ML.
Requires a large dataset that is labelled or categorised (referred to as the training set).
Classification is essential for ranking and detection.

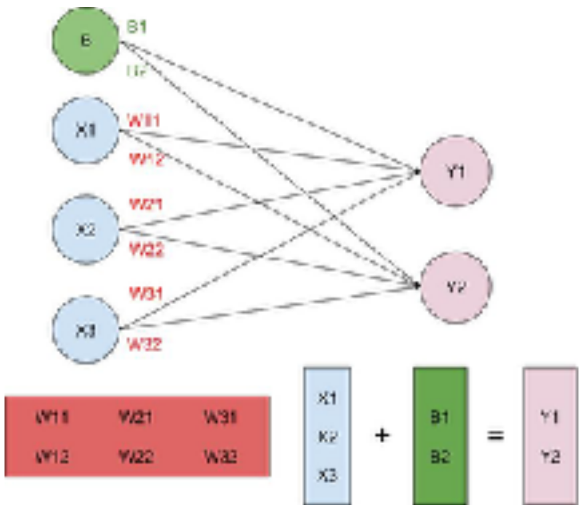
Logistic classifier
Logistic classifiers are referred to as linear classifiers

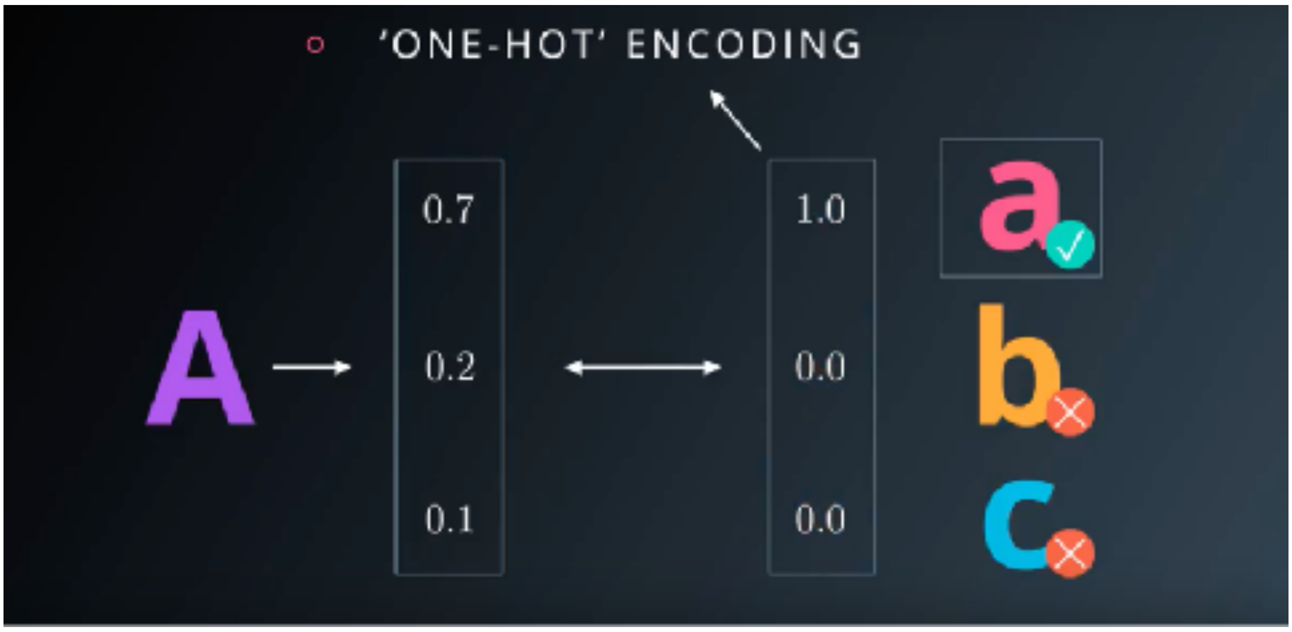
WX+b = y, y is the prediction, b is bias, X is input vector, W is the weight.



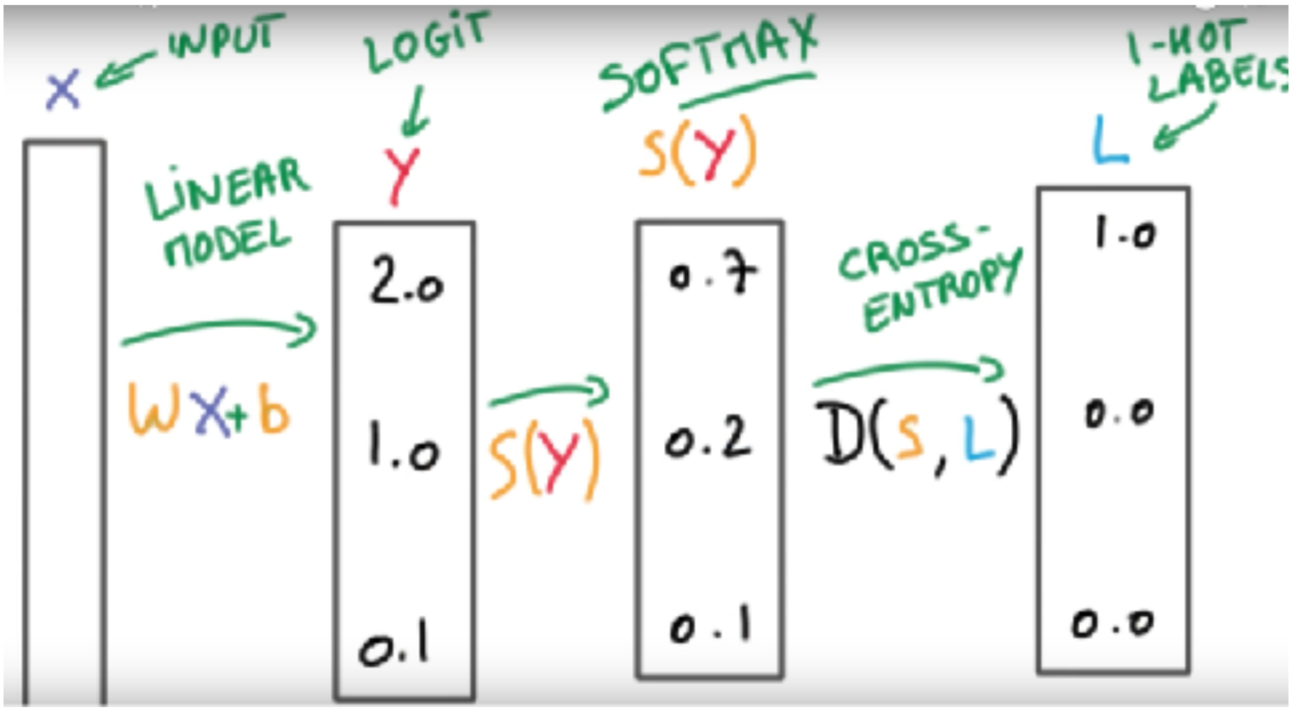


Forward Propagation





Current architecture is referred to as Multinomial Logistic Classification



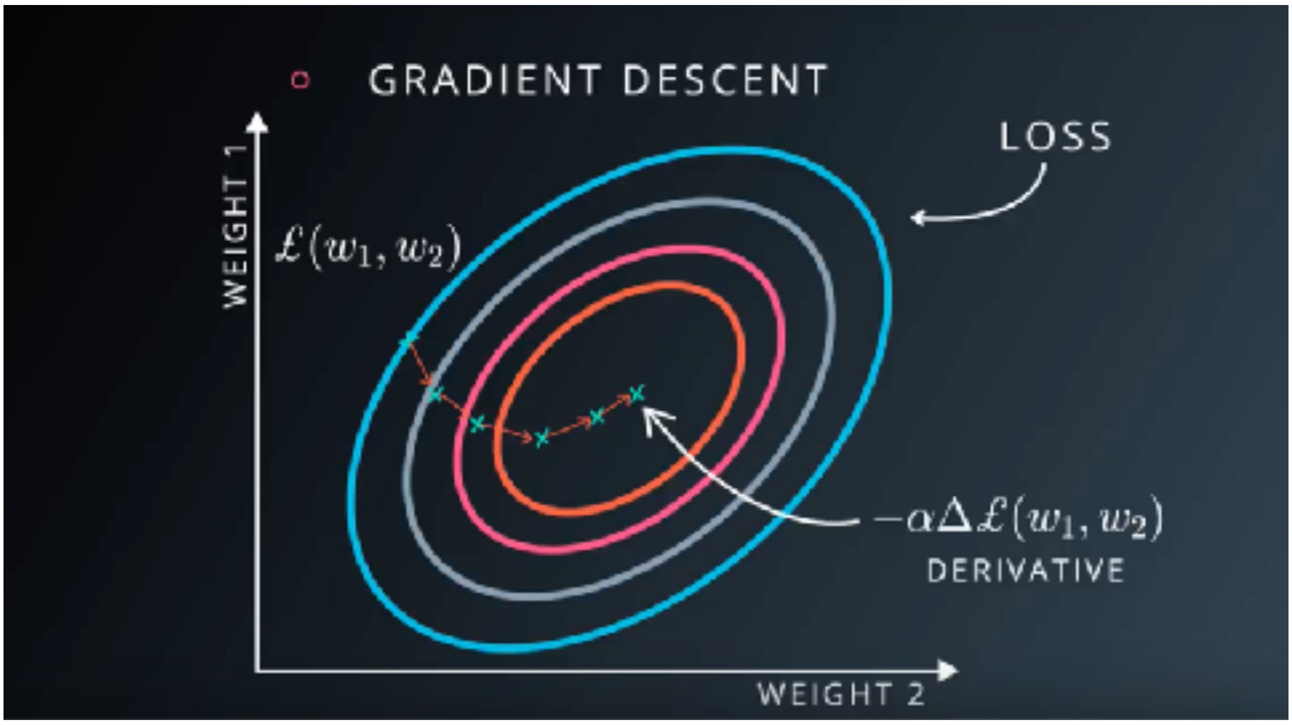
<u>Performance is measured as the average of CE for all datapoint and features, this is referred to as the training Loss.</u>

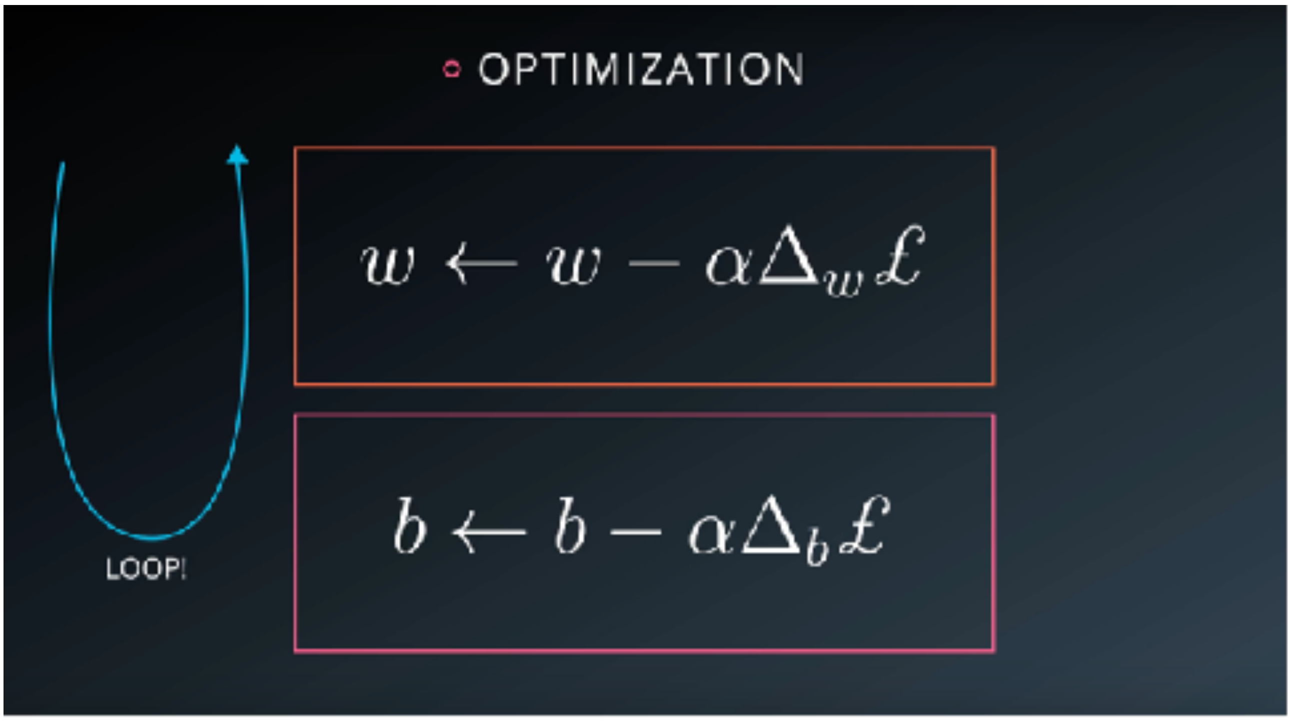
Classifiers will tend to memorise the training data set.
Another data subset is then required to asses the performance (test data).
The classifier is never trained with dataset, however, the test data will bleed into the data.
So, we can add a Validation dataset (usually 30,000 data points), before using the test dataset (untouched until the final version).

See **<u>cross validation</u>**: https://www.openml.org/a/estimation-procedures/1

Machine learning now becomes a numerical optimisation problem to minimise the loss function.

Gradient Descent
is used.




OPTIMIZATION
$w \leftarrow w - \alpha\Delta_w\pounds$
$b \leftarrow b - \alpha\Delta_b\pounds$
LOOP!

Stochastic Gradient Descent

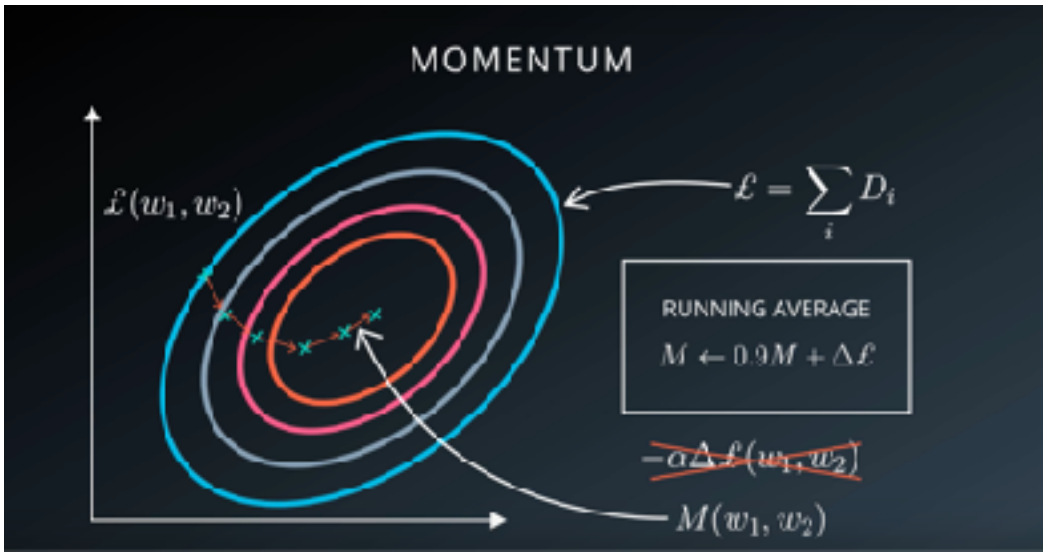
Gradient descent is a slow process for Cross Entropy, if CE takes n time to compute, GD will take 3xn, in this case CE calculates the error for each datapoint.

To improve the performance,

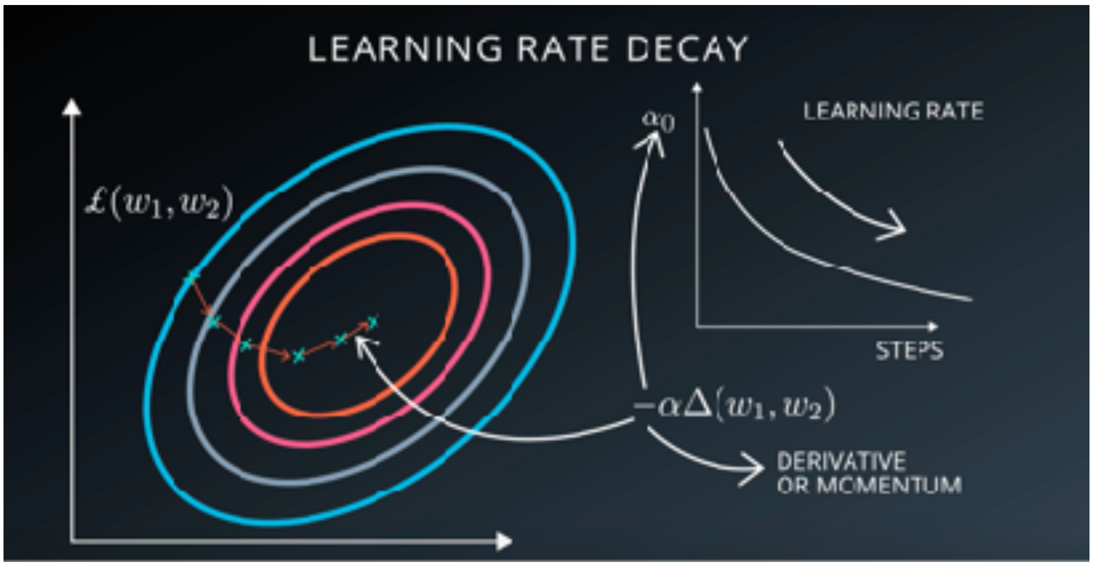
- the loss is calculated for an random subset of the data (1-1000).
- This is a bad metric, but it is quick and random!
    - So, increase iterations, & reducing learning rate (alpha)

To help SGD

- Zero centred inputs
- Zero mean, small variance initialisation for weights
- Use momentum, (running average of loss) instead using gradient



Learning rate decay (possible exponential decay or when plateauing).



Hyper parameters of SGD

1. Learning rate
2. Learning rate decay
3. Momentum
4. Batch size
5. Weight initialisation

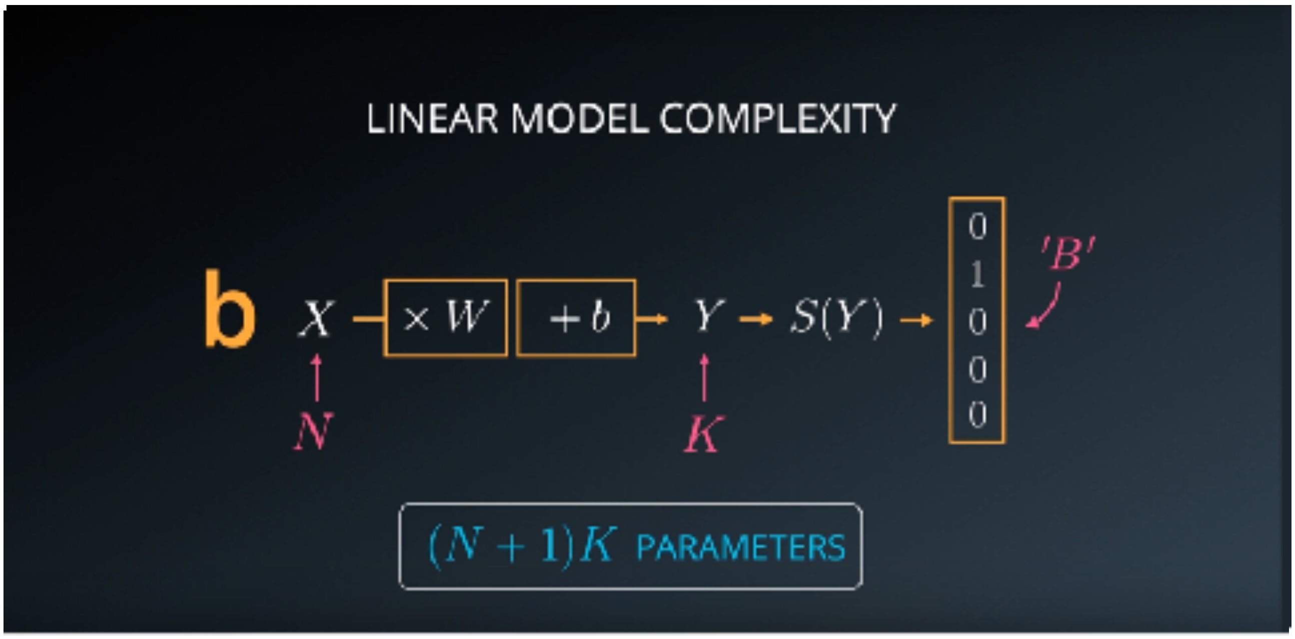
ADAGRAD and other algorithms attempt to automate the selection & tuning of those hyper parameters

http://ruder.io/optimizing-gradient-descent/index.html#adagrad

ADADELTA https://arxiv.org/pdf/1212.5701.pdf

Lesson 6: Introduction to Deep Neural Networks

Linear Classifier Complexity

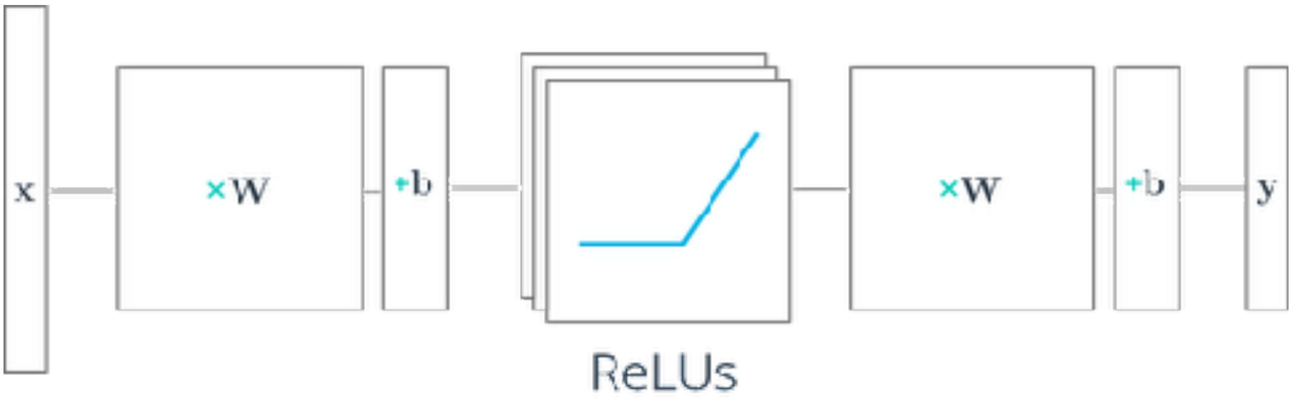


This is limited and complex

Linear functions are stable, easy to differentiate.

Rectified Linear Units (ReLUs)

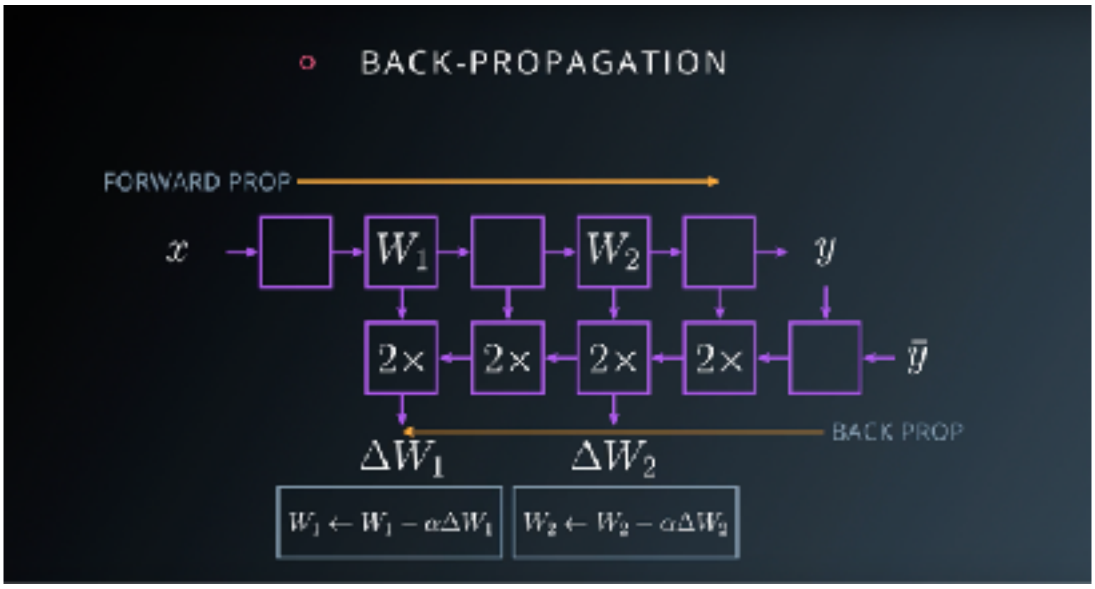
A Rectified linear unit (ReLU) is type of activation function that is defined as **f(x) = max(0, x)**. The function returns 0 if x is negative, otherwise it returns x. TensorFlow provides the ReLU function as **tf.nn.relu().** A derivative of a ReLU is a step function. using a non-linear activation function on the hidden layer lets it model non-linear functions.



1) The first layer effectively consists of the set of weights and biases applied to X and passed through ReLUs. The output of this layer is fed to the next one, but is not observable outside the network, hence it is known as a hidden layer.
2) The second layer consists of the weights and biases applied to these intermediate outputs, followed by the softmax function to generate probabilities.

Back prop

Used to calculate gradients for weights in the model.
Those gradients are used to modify the model using a learning rate.



Train Model and Save Training Parameters

Training a model can take hours. But once you close your TensorFlow session, you lose all the trained weights and biases.

```
# The file path to save the data
save_file = ‘./model.ckpt’
```

(The ".ckpt" extension stands for "checkpoint".)

```
# Class used to save and/or restore Tensor Variables
saver = tf.train.Saver()
```

```
 # Save the model
saver.save(sess, save_file)
print('Trained Model Saved.')
```

```
# Load the weights and bias
saver.restore(sess, save_file
```

You'll notice you still need to create the `weights` and `bias` Tensors in Python. The **`tf.train.Saver.restore()`** function loads the saved data into `weights` and `bias`. Since **`tf.train.Saver.restore()`** sets all the TensorFlow Variables, you don't need to call **`tf.global_variables_initializer()`**.

Instead of letting TensorFlow set the `name` property, let's set it manually to avoid assignment errors on load:

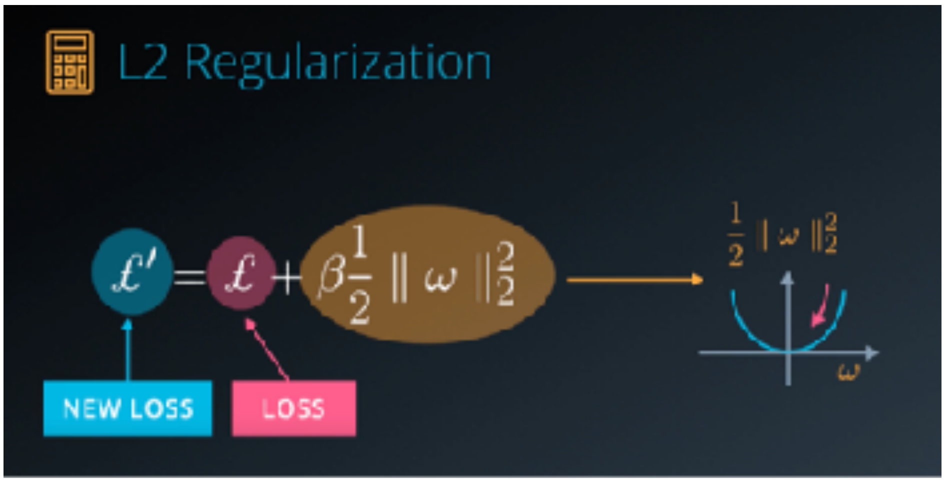
```
# Two Tensor Variables: weights and bias
weights = tf.Variable(tf.truncated_normal([2, 3]), name='weights_0')
bias = tf.Variable(tf.truncated_normal([3]), name=‘bias_0')
```

Regularisation

Use slightly larger networks than required for the given dataset then avoid over fitting. Regularisations are techniques used for overfitting.

a) **Early Termination**, stop as soon as we start fitting.

b) **L2 Regularisation**, is achieved by adding the L2 Norm of the weights to the cost function to avoid larger weights, hence avoiding over fitting (or over relying on a single parameter).

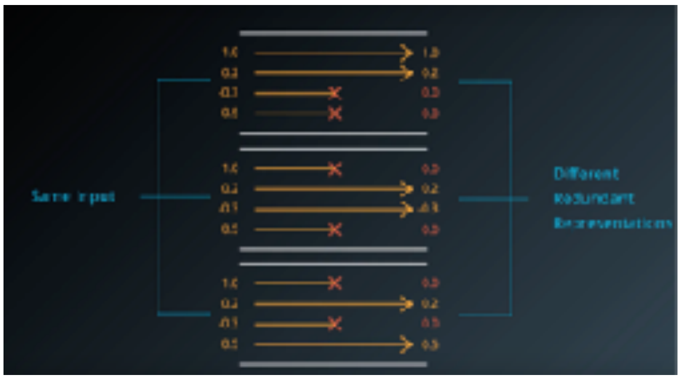

L2 Regularization
$\mathcal{L}' = \mathcal{L} + \beta\frac{1}{2}\|\omega\|_2^2$
$\frac{1}{2}\|\omega\|_2^2$
ω
NEW LOSS
LOSS

This is simple and can be easily calculated without modifying the network.

Derivative of L2 Norm is simply the vector of weights (w).

c) **Dropout**
Randomly disable activation so that the network does not rely on any activations and enables more redundant representations. (i.e. the solution is the consensus of different networks).

If dropout does not improve performance, a bigger network is recommended



Dropout should not be used in the evaluation stage after training. (Evaluation should be deterministic)
The evaluation is the average of the training labels. To maintain the scale, the labels should be scaled by the number of disabled activations.

"Dropout: A Simple Way to Prevent Neural Networks from Overfitting"
https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Tensor Flow Drop Out

The **`tf.nn.dropout()`** function takes in two parameters:

1. `hidden_layer`: the tensor to which you would like to apply dropout
2. `keep_prob`: the probability of keeping (i.e. *not* dropping) any given unit

`keep_prob` allows you to adjust the number of units to drop. In order to compensate for dropped units, **`tf.nn.dropout()`** multiplies all units that are kept (i.e. *not* dropped) by `1/keep_prob`.
-During training, a good starting value for `keep_prob` is `0.5`.
-During testing, use a `keep_prob` value of `1.0` to keep all units and maximize the power of the model.

Lesson 7: Convolutional Neural Networks

Using structure to teach the network.

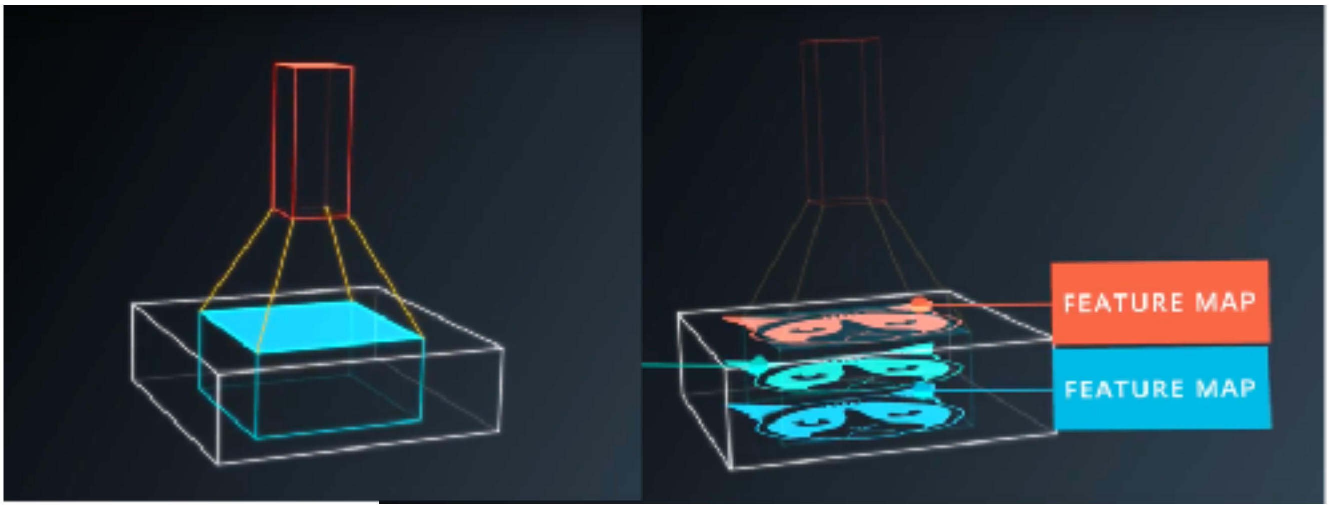
Statistical invariance: Telling the model that some parameters do not contribute to the classification (e.g. colour or position).

Weight sharing: When two inputs have the same label (same across time or space).
For images this uses CNN, for text, we use RNN.

COVNETS

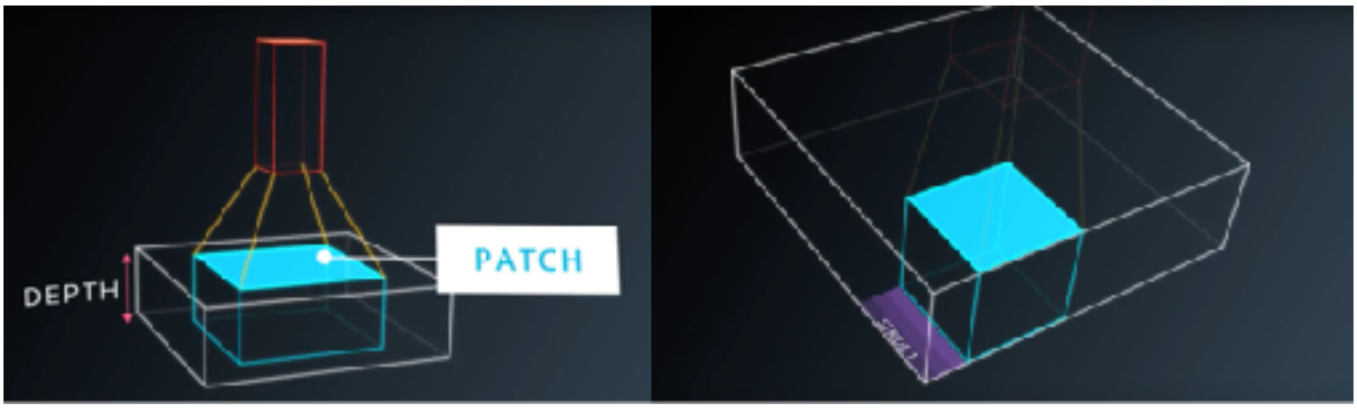
Neural networks that share their parameters across space. Such that, a single network can be translated using the same parameters across an image. Effectively, squeezing out spatial information and increasing the depth of information extracted from a single image.

A CNN might have several layers, and each layer might capture a different level in the hierarchy of objects. The first layer is the lowest level in the hierarchy, where the CNN generally classifies small parts of the image into simple shapes like horizontal and vertical lines and simple blobs of colors. The subsequent layers tend to be higher levels in the hierarchy and generally classify more complex ideas like shapes (combinations of lines), and eventually full objects like dogs.



For an image, the image stack depth is RGB (3), and a patch or a kernel is translated across the image.

Stride is the number of pixels that the filter is translated by. Stride of 1, results of output equal to input. Stride of 2 results in roughly half the size. Depending on what we do at the edge (same padding or valid padding).



Creating a CNN Filter

The first step for a CNN is to break up the image into smaller pieces. We do this by selecting a width and height that defines a filter.
The filter looks at small pieces, or patches, of the image. These patches are the same size as the filter.

The amount by which the filter slides is referred to as the 'stride'. **The stride is a hyperparameter** which, can be tuned. Increasing the stride reduces the size of the model by reducing the number of total patches each layer observes. However, this usually comes with a reduction in accuracy

What's important here is that we are grouping together adjacent pixels and treating them as a collective.

In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason and have special meaning.

By taking advantage of this local structure, our CNN learns to classify local patterns, like shapes and objects, in an image.

Filter Depth

It's common to have more than one filter. Different filters pick up different qualities of a patch. For example, one filter might look for a particular colour, while another might look for a kind of object of a specific shape. The amount of filters in a convolutional layer is called the filter depth.

Choosing a filter depth of k connects each patch to k neurons in the next layer. It is a tunable hyperparameter

Multiple neurons can be useful because a patch can have multiple interesting characteristics that we want to capture. Such that, each layer can capture different characteristics.

CONVETS Parameters

Sharing Weights
The weights and biases we learn for a given output layer are shared across all patches in a given input layer. Note that as we increase the depth of our filter, the number of weights and biases we have to learn still increases, as the weights aren't shared across the output channels. Sharing parameters not only helps us with translation invariance, but also gives us a smaller, more scalable model.

Dimensionality
our input layer has a width of **W** and a height of **H**
our convolutional layer has a filter size (edge size) **F**
we have a stride of **S**
a padding of **P**
and the number of filters **K**,
the following formula gives us the width of the next layer: **W_out =[(W−F+2P)/S] + 1.**
The output height would be **H_out = [(H-F+2P)/S] + 1**.
And the output depth would be equal to the number of filters **D_out = K**.
The output volume would be **W_out * H_out * D_out.**

Tensor Flow

```
input = tf.placeholder(tf.float32, (None, 32, 32, 3))
filter_weights = tf.Variable(tf.truncated_normal((8, 8, 3, 20))) # (height,
width, input_depth, output_depth)
filter_bias = tf.Variable(tf.zeros(20))
strides = [1, 2, 2, 1] # (batch, height, width, depth)
padding = 'SAME'
conv = tf.nn.conv2d(input, filter_weights, strides, padding) + filter_bias
```

Traditional NN
Parameter Size = (Filter Size + 1) * Output Layer

CNN
Parameter Size = (Filter Size + 1) * Output Layer Depth

https://www.tensorflow.org/api_guides/python/nn#Convolution

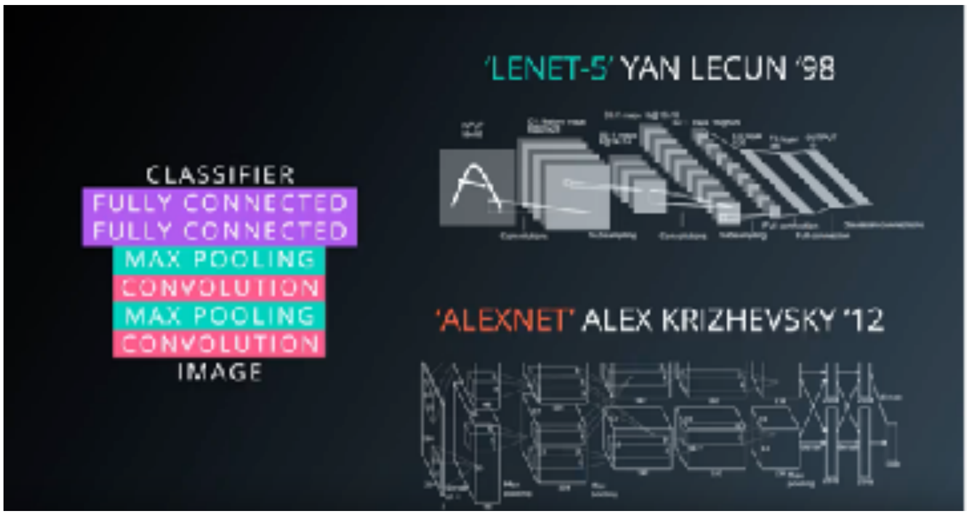
Pooling

Pooling is an approach to down sampling the image, without skipping samples.
For instance, max pooling, takes the maximum response from all networks.
Another approach is average pooling (i.e. blurring the feature map).

```
new_height = (input_height - filter_height)/S + 1
```

Famous architectures using Pooling

First usage of pooling.



Tensor Flow Pooling

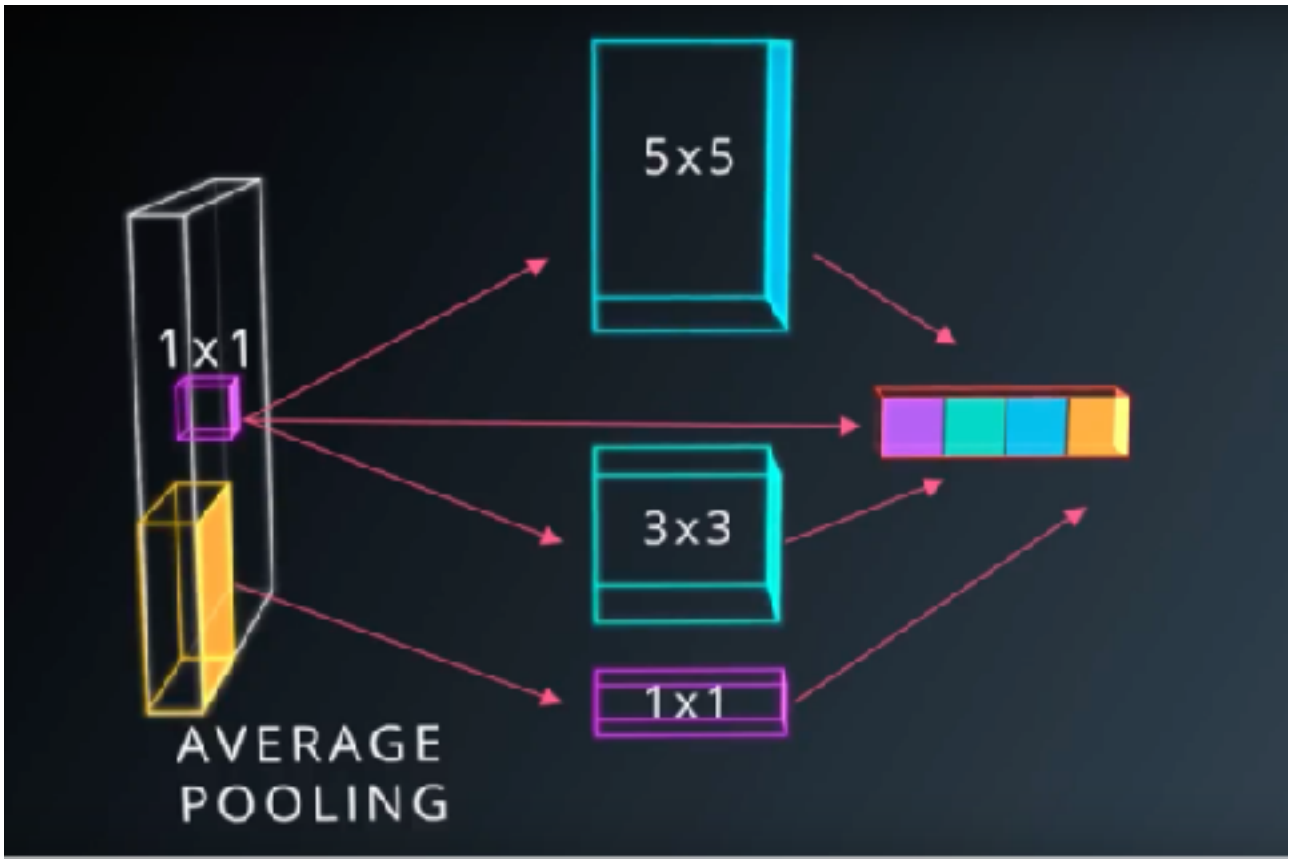
```
# Apply Max Pooling
conv_layer = tf.nn.max_pool(
    conv_layer,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')
```

The tf.nn.max_pool() function performs max pooling with the ksize parameter as the size of the filter and the strides parameter as the length of the stride. 2x2 filters with a stride of 2x2 are common in practice.

The ksize and strides parameters are structured as 4-element lists, with each element corresponding to a dimension of the input tensor ([batch, height, width, channels]). For both ksize and strides, the batch and channel dimensions are typically set to 1.

Inception

Instead of choosing between, 1x1, 3x3, 5x5 Conv nets, or pooling, we use all methods and concat output.



https://www.tensorflow.org/api_guides/python/nn#Convolution

Tensor Flow CNN Mechanics

'SAME' Padding

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
```

'Valid' Padding

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

Lesson 12: Keras

Keras is a high-level neural networks library, written in Python and capable of running on top of either TensorFlow or Theano

The keras.models.Sequential class is a wrapper for the neural network model. It provides common functions like fit(), evaluate(), and compile().

Layers

A Keras layer is just like a neural network layer. There are fully connected layers, max pool layers, and activation layers. You can add a layer to the model using the model's add() function

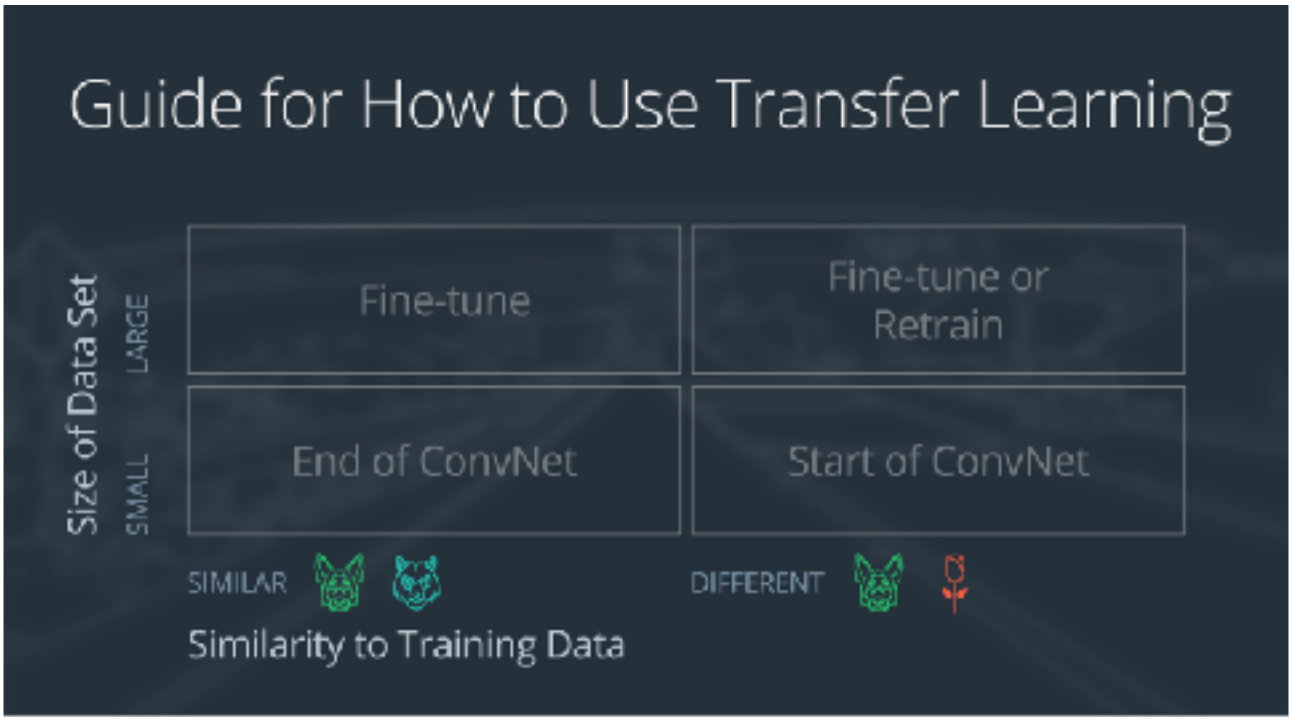
For instance, a CNN can be written as follow,

```
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(32, 32, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Dropout(0.5))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))
```

Lesson 13: Transfer Learning

Modifying deep neural networks to repurpose their application. The goal is to take advantage of networks that have already been trained instead of starting from scratch.

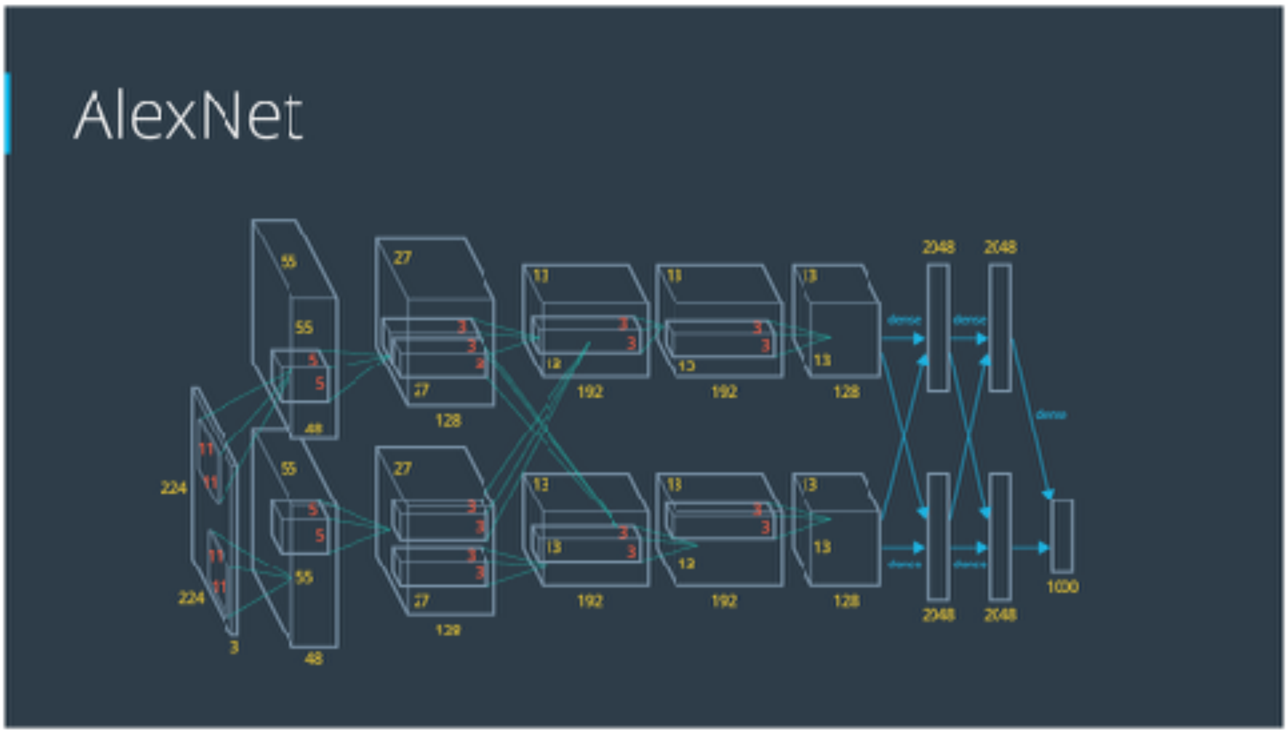
GPUs are ideal for NN training as they parallelise processes.



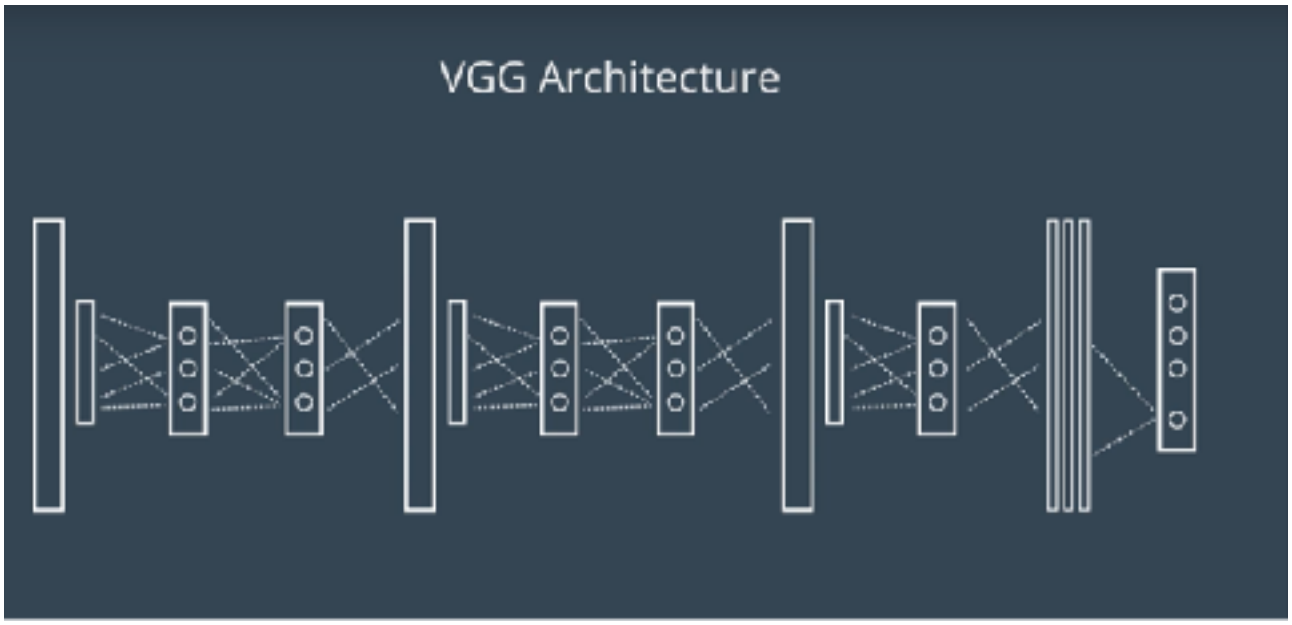
Size of Data	Similarity	Strategy	Steps
Small	Similar	Feature Extraction	Replace last connected network with new correct number of classes. Retrain Last Layer whilst rest network is frozen
Small	Different	Feature Extraction	Remove pre trained layers near the start Add connected layers to match number of class in new data Randomise weights in new layer Retrain Last Layer whilst rest network is frozen
Large	Similar	Fit Tune	Replace last connected network with new correct number of classes. Randomized new layer Retrain the entire network (starting from previous values)
Large	Different	Fit Tune / Retrain	Same as previous, if fails, randomize entire network weights and retrain.

ImageNet
A database of labelled images

AlexNet (2012)
Similar to LeNet, it has two networks running on two GPUs.
AlexNet pioneered Dropout and ReLu.

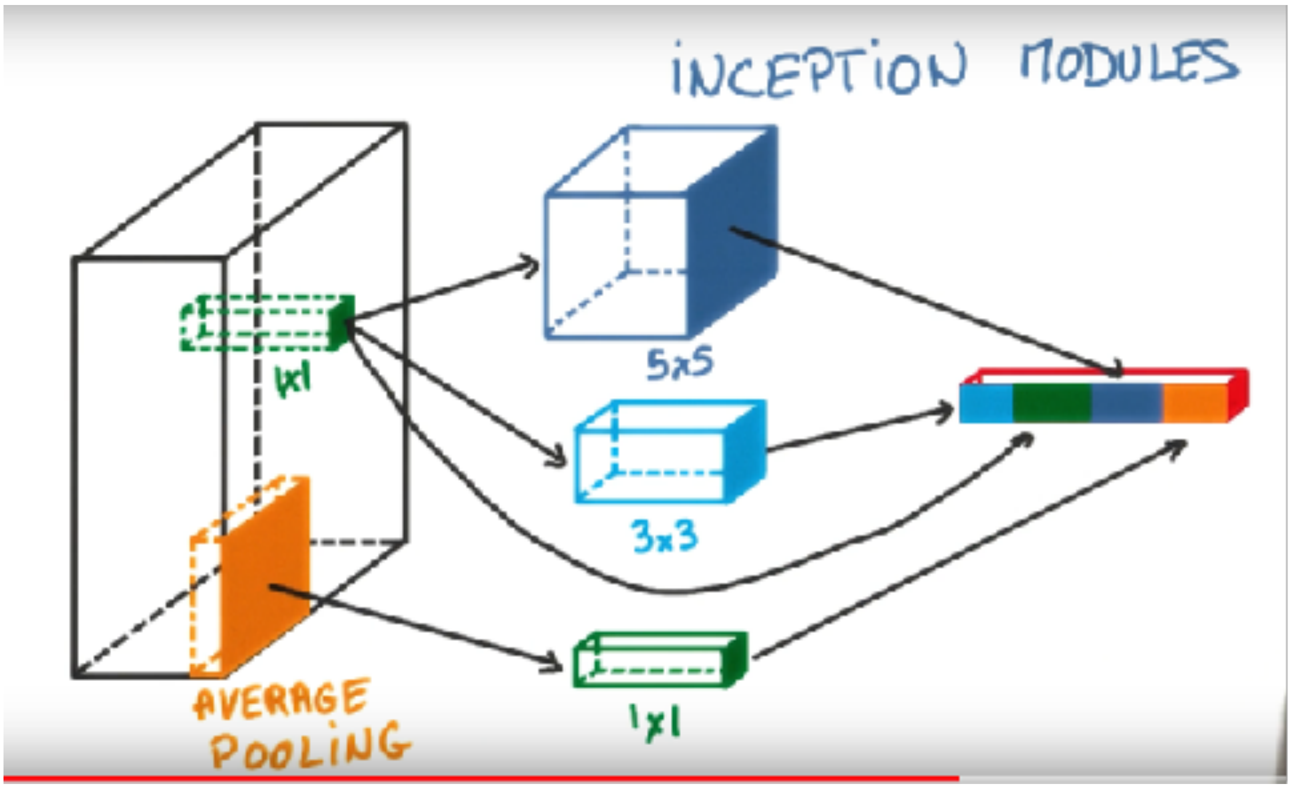


VGG (2014)
It is a series for 3x3 Convolutions followed by 2x2 pooling layers. It’s strength is its simplicity.



GoogLeNet (2014)

Pioneered Inception module
It concats, Pooling, Convolutional 1x1, 3x3, 5x3
Best option for running a network realtime



ResNet (2015)

Microsoft research, 150 layers.
Add connections to NN that skip layers to improve training.